March 2010

Fine-Grain Parallelism

Brian Tate
Worcester Polytechnic Institute

Christopher Pardy
Worcester Polytechnic Institute

Christopher Eduard Smith
Worcester Polytechnic Institute

Erik L. Archambault
Worcester Polytechnic Institute

John Forrest Hogeboom
Worcester Polytechnic Institute

See next page for additional authors

Author
Brian Tate, Christopher Pardy, Christopher Eduard Smith, Erik L. Archambault, John Forrest Hogeboom, and Joshua A. Montgomery

Fine-Grain Parallelism

An Investigative Study into the merits of Graphical Programming and a Fine-grain Execution Model

A Major Qualifying Project Report
Submitted to the Faculty of

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the
Degree of Bachelor of Science
By

Erik Archambault *John Hogeboom* *Joshua Montgomery*

Christopher Pardy *Christopher Smith* *Brian Tate*

Advised By:

Professor Craig E. Wills

Professor Hugh C. Lauer

March 2010 MQP-CEW-1002

Abstract

Computer hardware is at the beginning of the multi-core revolution. While hardware at the commodity level is capable of running concurrent software, most software does not take advantage of this fact because parallel software development is difficult. This project addressed potential remedies to these difficulties by investigating graphical programming and fine-grain parallelism. A prototype system taking advantage of both of these concepts was implemented and evaluated in terms of real-world applications.

Contents

Executive Summary

A steady increase of performance is expected by the computing industry. For years, taking advantage of increasing chip density and correlated increasing clock speeds, new chips were engineered over the years which maintained a steady increase in performance. In order for computing performance to keep increasing into the next decade, chip designers began to create processors that supported parallel computing. While hardware support exists for concurrency, current software abstractions have not been adequate to promote a proliferation of parallel software. This project set out to investigate ways to enhance the programmer's ability to construct parallel applications. Ultimately, a system that utilized the concepts of graphical programming and fine-grain parallelism was implemented.

The first step in our project was a thorough investigation of data flow languages. Dataflow languages have a structure that is more appropriate for a fine-grain execution model, because they are based on the execution of blocks with inputs and outputs. This translates into the notion of fine-grain parallelism, which is best summarized as the massive parallel execution of a number of immutable execution blocks. Each execution block is queued and executes when its inputs are available. When a block finishes it publishes its outputs, making them available for future blocks. Through the analysis of dataflow languages, we found SISAL was a good fit as a foundation for our project. Using SISAL as a base we began the task of creating a graphical programming front end and a runtime utilizing fine-grain parallelism. The overall architecture of our prototype system involved three phases to construct a program. The first phase was creating a visual graphical representation of a program and converting it into SISAL code. Using the SISAL compiler we convert the code into IF1, which is a language based on the concept of directed cyclic graphs. In second phase, our program converts the IF1 into a C++ representation compatible with our runtime library. Our runtime library implements the thread, memory, and block management for fine-grain parallelism. Using a standard C++ compiler we compile the output from the second phase with our runtime library to create a standard threaded executable.

Upon finishing the construction of the system, we evaluated it by investigating the feasibility of implementing real-world algorithms using both the graphical front end and fine-grain parallelism concept. We examined the Fast Fourier Transform, H.264 video codec, and the Quicksort sorting algorithm. All three were evaluated in terms of graphical programming and feasibility in terms of a fine-grain execution model. All three were found to have popular 3rd-party graphical representations, and these diagrams helped express concurrency. A fine-grain execution model also allowed for an increase in the maximum level of concurrency for each individual algorithm.

Ultimately, our project's results show that fine-grain parallelism and the use of graphical programming show promise for parallel computing. By using the concepts we investigated and developed a system for, real gains in ease of writing parallel code could be achieved. As hardware designs are heading in a parallel direction for the future, it is vital that software research keeps pace and tracks those changes. Future work should be built upon the system we developed, or the concepts we investigated and formulated in order to maximize the usefulness of parallel computing hardware.

1 Introduction

Computing technology has increased in performance and commercial adoption unparalleled to any other industry. Computing has been able to penetrate most commercial markets including home appliances, automobiles, and communication due to the promise of solving recurring complex problems easily without expensive custom hardware. While computing chips in toasters don't have the same performance requirements of those in communication devices or personal computers, the trend is to increase performance of a chip, lowering execution times. Moore's Law has correctly predicted the number of transistors on a cost effect integrated circuit doubles every two years. Companies like Intel and AMD used these extra transistors to allow performance gains to track this exponential growth on their developed processors. They also were able to increase clock rates because of the design of these ultra-dense integrated circuits. Unfortunately, increasing clock speed also increases heat and power consumption causing further increases in clock frequency to require significant power source and heat dissipation technology. Thus, to attempt increases in performance, chip manufacturers have had to switch to using the increasing number of transistors on chip components that implement concurrency rather than higher clock speeds.

Hardware designers have led software designers in the world of parallel computing. Initial parallel computers focused on mounting multiple processors on a single motherboard. These machines shared a common memory for each processor, but each processor was a full-fledged CPU with its own cache hierarchy. Recently hardware architectures have shifted to allow for multiple processing cores on a single processor. The design decision to focus on multi-core architectures has improved parallel processor performance because of shared caches. This smaller silicon footprint has allowed cheaper commodity hardware to run concurrent software faster. Although many modern processors are designed for concurrency, implementing concurrency in software is still a difficult task. Complex race conditions, lack of a good representation, and increased program complexity all plague parallel software development. One of the most important challenges for the computing community is to make programming concurrent systems more straightforward, reliable, and testable.

There has been significant progress increasing support for concurrency in scientific software, but much of it focuses towards loop-level parallelism where gains are seen in iterations over a large set of data. Existing tools such as OpenMP and Thread Building Blocks all focus on this area of concurrency. Programmers use these tools to explicitly implement parallelism at a course-grain level. While it saves a programmer from personally managing threads and locking it still requires some knowledge of them and limits the parallelism to loop-bound operations. Implicit parallelism would remove these limits, allowing a programmer to implement a concurrent application without having to be an expert on threads, locks, or race conditions. Fine-grained parallelism is the idea of breaking up a large serial computation into a group of smaller blocks that may be run in parallel, at a level at least as fine as function calls or finer. Combining implicit parallelization and fine-grained parallelism would be a powerful tool. A complete system would need to adequately compute the basic blocks for a computation. This problem is analogous to compilers; can a computer beat a human at optimizing computer code? For compilers the answer has been resounding positive. The use of compilers decrease development costs, maintenance

costs, reduces errors and allows a programmer to represent problems in addressable ways. A similar leap made in the area of parallel computing would be a major step forward in the multi-core revolution.

Despite there being a new pressure for researchers and industry to find better solutions to concurrent software challenges, there hasn't been a strong focus on implicit parallelism. This is potentially due to the paradigm change that would need to occur in concurrent software languages. Implicit parallelism leaves implementation details to automatic processes, which depend on a constrained dataflow. Therefore popular languages like C, C++, or Java, which are loosely constrained, cannot be accurately translated to use implicit parallelism. If the advantages of implicit parallelism are to be realized, the inherently serial languages of the past three decades would need to be abandoned.

Another issue is the effect of how programs are represented. While the majority of programming languages are entirely textual, there are graphical languages as well. Thus there also exists the unanswered question of whether graphical representations can express concurrency better than traditional textual representations of programs. Besides a few specialized programming languages, graphical languages have been largely rejected by mainstream software engineering. Thus there has been little research on the effectiveness of graphical programming languages for concurrency. Compounded with the idea of fine-grained implicit parallelism as a run-time model for concurrency a graphical programming language is also an idea worth exploring.

This project set out on a two-prong exploration into both graphical programming and fine-grain parallelism efforts. We began by researching existing techniques to enhance parallel computing. This investigation led us to dataflow languages; these languages have a model that is seemingly a better fit for parallelism than procedural languages. After completing this background research, the project changed focus to implement a prototype environment for graphically programming a software system that ran in a developed fine-grain runtime model. Once the system was completed, effectiveness needed to be measured in terms of amount of parallelism extracted both theoretically and empirically. Ultimately, by completing this analysis we found there to be a use to exploring fine-grain parallelism further. As predicted, this runtime model seemingly extracts hidden parallelism in a problem. While not as conclusive, we also found that while a graphical representation of a problem can get complex if not represented properly, it does allow the entire problem to be seen at once by a programmer. This allows big picture optimizations, including parallelism, to be spotted.

For an overview of our background research into dataflow languages and current efforts to enhance parallel computing, refer to Chapter 3. In Chapter 4 we present an overview of our entire architectural vision for the project and formalize the design of fine-grain parallelism. Chapters 5, 6, and 7 describe the graphical front-end, runtime library, and code generation efforts in more detail, respectively. Chapter 8 evaluates the notion of fine-grain parallelism for a variety of real-life algorithms including the Fast Fourier Transform, Quicksort, and the h.264 video codec. Chapter 9 concludes our report, summarizes the evaluation of the big picture, and discusses the overall successes and failures of our system.

2 Background

In this chapter the background information that provides the context for this project is presented. This includes discussion of parallelism in computation, including its history, its applications, the challenges it presents, and the potential benefits it offers. This background also includes discussion of the dataflow programming languages we examined to determine what has been achieved in terms of exploiting implicit opportunities for parallelism. Three languages are addressed in particular, LabVIEW, the language that served to inspire this project, Oz/Mozart, a language we investigated for its potential use in this project, and SISAL, the language that we make considerable use of in this project.

2.1 Parallelism

Parallelism, being able to execute multiple parts of a program concurrently, is becoming increasingly important. This change is due to the move of processor manufacturers away from their traditional methods of making faster individual processor cores to the new trend of making processor chips with more processing cores on them.

2.1.1 Examples of Parallel Computations

Due to the previous manufacturer trend of increasing the speed in single core processors, serious applications of parallel computation have long been solely the domain of high performance computing, because true parallel computation cannot take place on the traditional single core processor. Scientific computing is an area where high performance computing is a practical necessity. This need is due to due to the computations being so complex and involving heavy number crunching to the degree that they would take a prohibitively long time unless the problem could be broken down into pieces that could be computed concurrently. Parallelism has also long been employed by specialized hardware for specific applications, such as digital signal processing and graphics processing.

Due to the fact that scientific computing is an area where the programs are computationally intensive and the algorithms involved often offer opportunities for parallelism, it is a primary area where the application of parallelism is not only employed but actively researched. A 2006 paper on the landscape of parallel computing research [1] gives 13 general classes of computation and algorithms that are expected to be important to science and engineering for some years to come. The area of scientific computing stands to benefit from any speedups due to parallelism achievable in these classes of computations. These classes are, along with some examples of their applications:

-Dense Linear Algebra (MATLAB, H.264 video compression)

-Sparse Linear Algebra (Computational fluid dynamics, linear program solver)

-Spectral Methods (Fast Fourier Transform [FFT])

-N-Body Methods (Molecular Dynamics)

-Structured Grids (MPEG encoding/decoding, smoothing and interpolation of graphics)

-Unstructured Grids (Belief propagation in machine learning)

-MapReduce (Ray tracing, efficiently processing large databases)

-Combinational Logic (Encryption, hashing)

-Graph Traversal (Playing chess, Bayesian networks, natural language processing)

-Dynamic Programming (Playing go, database query optimization)

-Backtrack and Branch-and-Bound (Playing chess, constraint satisfaction)

-Graphical Models (Viterbi decoding, Hidden Markov models)

-Finite State Machine (MPEG encoding/decoding, bzip2 compression, H.264 video compression)

Among these examples of parallel computation, the algorithms for Fast Fourier Transform and H.264 video encoding are discussed in detail in Chapter 7. This provides a look into the algorithms and the opportunities for parallelism therein, with a focus on the type of fine-grained parallelism this project is about.

2.1.2 History of Parallel Abstractions

In the early days of computing, computers would execute a single job, i.e. a task or program, to completion, with exclusive access to the processor from the beginning of the job to its end. This method of handling jobs strictly one after another is inefficient, because the processor is not being utilized whenever a particular job has to wait on some external factor, such as an input/output device. At least as early as the late 1950s, mainframes were in use, necessitating time sharing of a processor among the jobs of multiple users [2]. However, machines at the time still handled jobs in sequential batches, rather than sharing the processor among jobs [3]. It was not until the early 1960s and the advent of virtual memory that processes and context switching came to be more or less as they are known today [4].

Since that time, or the beginning of multiprogramming, *processes* have been the primary abstraction for independent units of code which conceptually, if not literally, execute in parallel. Processes which have their own independent context and the ability of the operating system and processor to switch between these processes and their contexts allow efficient utilization of the processor. However, context switching is expensive relative to other costs of processing, such as single operations or even substantial sequences of operations. Thus, more “lightweight” abstractions have been developed, which do not incur such high overhead.

The first of these newer parallel abstractions are *threads*. Threads essentially exist within, or on top of, processes, so that a switch can occur between threads executing without a more expensive context switch between processes. Modern processors even support threads to a degree in the hardware, which helps make them efficient [5]. Aside from threads being more lightweight and efficient, the main difference between threads and processes is that threads within a given process share the resources and context of that process, while processes have independent resources and context. This can have benefits, such as easy communication between threads, and drawbacks, such as those discussed below in Problems with Parallelism.

Although the use of threads provides even more possibilities and flexibility for parallel programming, the "context" switching of threads is still a cost it would be desirable to avoid. As a result, the concept of *fibers* has come about. Fibers are intended to be even more lightweight than threads. They are essentially user-level threads that are implemented on top of actual threads [6]. In this way, fibers are related to threads in a similar way to how threads are related to processes, except that processes and threads are operating system-level constructs whereas fibers are at the user level. Any switching between fibers is thus handled at the user level and any switching costs at the operating system/processor level like those associated with switching between processes and threads can be avoided. Since fibers are even lighter-weight, user-level constructs, the benefits and drawbacks of threads mentioned above can be even greater with fibers.

2.1.3 Problems with Parallelism

Being able to exploit implicit parallelism in a programming language is desirable largely because of the problems that can result when programming in an explicitly parallel manner. Implicit parallelism allows the programmer to achieve parallel execution while still thinking in a more or less sequential programming way. It is easier for programmers to implement algorithms and such in this way, rather than having to also deal explicitly with concurrency issues such as those that follow:

2.1.3.1 Race Conditions

A race condition occurs when the result or state of a program is, usually unintentionally, dependent on timing. This dependence affects the order of execution of operations that can run concurrently. Race conditions are often the result of some shared resource, such as a memory location, being accessed by multiple concurrent agents, such as processes or threads. Thus, using some form of mutual exclusion can usually prevent race conditions. This ensures that when a concurrent agent is in its critical region, i.e. the part of its code where it makes use of a shared resource, it has exclusive access to the resource so that other agents may not use it before the current user is finished.

While race conditions can be avoided in explicitly concurrent code, they still pose problems. Mutual exclusion adds overhead to both the act of programming as well as the runtime execution of the code. Race conditions can also be difficult to debug since they are timing dependent. A program with a race condition bug can run flawlessly or crash at different times, depending on how the concurrent parts of the program get scheduled by the operating system. As a result, taking this burden off of the programmer and handling it automatically when the code is compiled facilitates the production of bug-free concurrent programs.

2.1.3.2 Code Complexity

As exemplified by the discussion of race conditions above, creating correct, explicitly parallel code increases the complexity of the code. Some commonly used languages that support explicit parallelism but not implicit parallelism are C, C++, and Java. In these languages, it is possible to create new processes and threads to execute concurrently. The mere creation of these concurrent agents along with their management increases the amount and complexity of code. Of course, there is also the issue of race conditions, for which mutual exclusion or some other precaution must be taken. Also, the programmer must make decisions about how to break a program down in concurrent pieces in order to

achieve efficiency and correct execution. The consideration of efficiency in particular can result in code that is clever, yet difficult to understand and debug.

There are also libraries, such as OpenMP [7], which can ease the task of parallel programming. In the case of OpenMP at least, the handling of process/thread creation and management can be simplified. However, the code is must still be made more complex to a degree, and the programmer must still ultimately decide how to parallelize the program.

Once again, it would seem that offloading the responsibility of parallelizing the execution of a program from the programmers to the language implementers would be conducive to the creation of correct, manageable concurrent code. Programmers would be able to focus on the core logic of the program rather than having to implement and debug the additional logic dealing with concurrency.

2.1.3.3 Asynchronous, Non-deterministic vs. Synchronous, Deterministic Execution

Traditionally, the vast majority of programs have been created to execute in a synchronous, deterministic manner. Determinism means here that over time the state of the program follows a clear causal path. At any point, the current program state will depend solely on the previous state, and will entirely determine the next state. Most importantly, this means program behavior is consistently reproducible, and a deterministic program will always follow the same path when all variables are the same. Synchronous means here that the execution of the program follows a predictable order where instructions always execute in the same relative order. The flow of control in the synchronous program is such that when control is passed to a function or subroutine, that part of the program will execute immediately, and the part of the program that transferred control will resume as soon as that function or subroutine is finished.

This behavior is in contrast to concurrent programs, which execute in an asynchronous, non-deterministic manner. Non-determinism means here that the state of the program as whole is not strictly dependent on the previous state of the program. Since there are multiple active agents comprising the program, e.g. processes, threads, or fibers, a given agent can execute faster or slower than others, or it can be halted and resumed due to operation system scheduling. These possibilities mean that the state of the program as a whole cannot be predicted based on its previous state, and cannot predict the next state. This does not mean that the execution of a single agent cannot be deterministic, but multiple agents executing concurrently and asynchronously leads to overall non-determinism.

Asynchronous means here that the different agents execute different parts of the program in an unsynchronized manner with no strict relative ordering. Each agent is to run in accordance with how the operating system schedules it, and the program itself cannot control this. This unpredictable scheduling is what leads to non-deterministic behavior. However, programming constructs can be used to force synchronization of the agents. Depending on the extent to which this is done, a program with concurrent agents can take on nearly or fully synchronous, deterministic behavior, but this would result in the loss of any gains to be had from parallel execution.

The asynchronous, non-deterministic behavior of concurrent programs is the root of most, if not all, logic bugs in such programs. Programmers tend to think about the execution of programs sequentially, where the program is synchronous and deterministic. Even when anticipating that programs that will not be executed sequentially, it is easy to miss a potentially problematic portion of code or to make basic assumptions about timing and synchronization that ostensibly seem reasonable but prove to be wrong in practice. Once again, this leads to the conclusion that allowing programmers to avoid dealing with these issues explicitly by handling them implicitly in the language can make it easier to get concurrency right.

2.1.4 Performance Increases due to Parallelism

Performance increases in parallel computing are not always linear. In computing, Amdahl's Law predicts the amount of performance gain in a situation with diminishing returns [8]. This situation occurs when a portion of the program is affected by the change, but another portion is not. One example of this is lowering disk access times. Performance of an application would only increase by a percentage related to the percentage of time spent doing disk I/O. The same logic applies for parallelism. Despite increasing the number of available processors, an application can only increase in performance relative to its percent of parallelizability [9].

$$New\ Execution\ time = \frac{Execution\ time\ affected}{Amount\ of\ improvement} + Execution\ time\ unaffected$$

Equation 2.1 – Amdahl's Law for Computing [9]

For parallel computing this can be expanded into a specialized case described in Equation 2.2.

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}}$$

Equation 2.2 – Amdahl's Law for Parallel computing. P is Percent of Parallelizable execution time, N is the number of Processors

Asymptotic behavior of this equation reveals $S = \frac{1}{(1-P)}$ as $N \rightarrow \infty$ because $\lim_{N \rightarrow \infty} \frac{P}{N} = 0$. This highlights the importance of maximizing the amount of parallelizability in a process. Because even if the number of processes on a system became huge, if the percent of parallelization is 50%, the maximum speedup you can get is 2x the serial performance. Graphically this rule of diminishing returns is displayed in Figure 2.1.

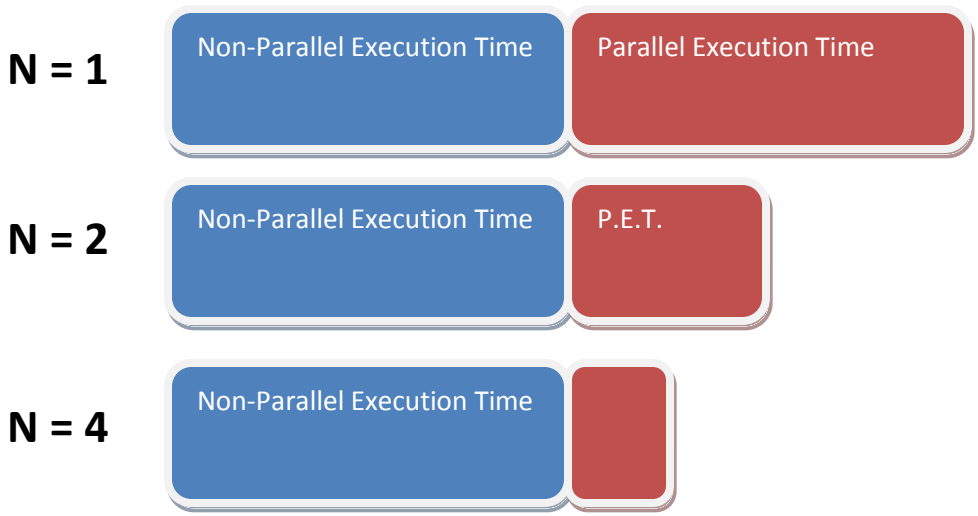

Figure 2.1 - Graphical representation of the diminishing returns of adding parallel processing units.

2.2 Dataflow Language Overview

Dataflow languages are distinguished by the fact that their syntax and general programming style is structured around the flow of data between different parts or "nodes" of a program. Any true dataflow program can be modeled in terms of a graph of these nodes connected by arcs or edges along which data flow. This perspective and modeling of a program as a graph is most strongly reinforced by graphical dataflow languages, where these nodes and their interconnections are clearly visible. Though even in textual dataflow languages like SISAL the way in which the program could be represented as a graph can often be inferred [10].

As there are a large number of dataflow languages in the world, we decided it would be beneficial to investigate a wide variety of these languages to expose ourselves to some of the methods and tools we could employ to reach our goals.

2.2.1 Binary Modular Dataflow Machine

BMDFM [11] was designed with goals similar to those we hope to achieve in this project: it focuses on producing speedup in programs by exploiting implicit concurrency, and it uses a fine-grain execution model with dynamic scheduling. The language itself is textual/sequential and C/C++-like, and it can be extended by user-defined functions written in C/C++. It is an academic project and the implementation is available free, but it is not open source and under a nonstandard license. Therefore, while this language is a great example of what we would like to accomplish, it is not particularly useful in helping us reach our goals.

2.2.2 Pro-graph

Pro-graph [12] is a graphical language that was created as an investigation into dataflow languages in the early 1980s. Commercial development environments for it were available into the late '90s, at which point it more or less faded into obscurity. It is a fairly standard graphical dataflow language with seemingly no unusual or notable features in comparison to other such languages. A new

commercial implementation has become available exclusively for Mac OS X in the form of the Marten IDE [13].

2.2.3 Oz

Oz [14] is a textual language developed for programming language education. It's design focuses on a simple syntax that can be expanded through the use of syntactic sugar. Its primary implementation is the Mozart Programming System, released with an open-source license. It has been ported to Unix/Linux, Windows, and Mac OS X. It was designed to be a concurrency-oriented language, meaning it should make concurrency easy to use and efficient. It features dataflow variables and declarative concurrency. Its design also allows for a network-transparent distributed computing model.

2.2.4 Simulink

Simulink [15] is a commercial tool for modeling dynamic systems. It features a graphical diagramming interface. It is related to the MATLAB environment. It is designed largely for signal, image, and video processing. There are a number of products designed to be used in conjunction with Simulink for applications such as developing state machines and flowcharts, as well as generating C, VHDL, and Verilog code for various purposes.

2.2.5 SISAL

SISAL [16] is a general-purpose textual dataflow language. It features implicit parallelism and Pascal-like syntax. Compilation involves production of a dataflow graph in Intermediary Form 1. An implementation is available on the SISAL project page [17], which states that the optimizing compiler gives high performance on commodity SMP architectures.

2.2.6 VHDL

VHDL [18] was originally based off of ADA and developed on the request on the US Department of Defense. The language is commonly used for circuit design. This means that the concept of wires and objects (such as gates) are already part of the language and could be used by other graphic packages. The language is still being improved and is regularly used for chip design. There is also a vast support of compilers available for VHDL.

2.2.7 PureData

PureData [19] is an open source graphical programming language released under a license similar to the BSD license. Pd is modular code based using either externals or objects, which serve as the building blocks for written software. The language can be extended in various languages such as C, Python, Ruby or Scheme. Pd is designed to support live collaboration. The language is more geared toward audio, but supports most of the primitive stuff such as mathematical, logical, and bitwise operators.

2.2.8 OpenWire

OpenWire [20] is an open source language and supplies examples of open source editors for Openwire. Openwire is stream based with components that have input and output streams. There are many applications of Openwire such as video processing, instrumental processing, signal processing, and vision processing. It seems to support a wide variety of things unlike a lot of the other data flow languages.

2.2.9 Max

Max [21] is a dataflow language that is geared toward music and multimedia development. Max is highly modular, with most of its routines existing in the form of shared libraries. Third-party development is allowed through the use of an API, which developers can use to develop external objects. External objects can be written in C, C++, JAVA, or JavaScript.

2.2.10 Cantata

Cantata [22] is a graphical dataflow language targeting image manipulation. Cantata is based around a suite of image manipulation tools called Khoros. These tools can be represented by Cantanta's visual representation of dataflow. Developers connect the Khoros sub-components into a network of connected nodes. The dataflow is represented in the traditional connections between nodes. Besides just image processing, it seems Khoros also does signal processing.

2.2.11 jMax

Focused on audio editing, this language and tool use terms synonymous with traditional DJ setups. Instead of nodes and edges, jMax [23] programmers use “patches” and “patch cords”, respectively. Audio manipulation seems like a good application for data flow languages as easy to comprehend stages like amplification, reverb/distortion/manipulation, and output.

2.2.12 Lucid

Lucid [24] is a textual language designed to experiment with non-von Neumann programming models. The programming model is based on an algebra of histories, or infinite sequences of data items. The implications of this include: variables represent infinite streams of values and functions serve as filters. Statements can be considered to define networks of processors and lines of communication through which data flow.

2.3 Dataflow Languages in Depth

The following languages, with the exception of LabVIEW, were investigated more in-depth as a result of our initial overview of the languages. LabVIEW is relevant because it inspired this project. The other languages, Mozart/Oz and SISAL, were deemed potentially useful to the project.

2.3.1 LabVIEW

LabVIEW [25] was not investigated as were the other languages here, but it is discussed because it is the language that inspired this project. LabVIEW is a graphical dataflow language designed for implementing virtual instrumentation. It was observed that by looking at graphical LabVIEW programs, opportunities for parallelism appear to “jump out” at the viewer. This observation is a result of the visualization of the flow of data through the program and its apparent path of execution. It is because of this it was hypothesized that it would be possible to extract implicit parallelism from the graphical representation of a program when it was compiled. Intuitively, the information that leads a person to see opportunities for parallelism is there, and the question is whether or not these opportunities could be detected and exploited by the compiler. Thus came about the overall goal of this project, which is to create a framework with a front-end graphical interface inspired by LabVIEW that is capable of compiling graphical programs into executables that are automatically parallelized.

One of this project's advisors, a user of LabVIEW, noted that in recent years its makers, National Instruments, have added the capability of exploiting some degree of implicit parallelism in LabVIEW programs, being able to scale the program to a small number of processors. However, it was also noted that LabVIEW was not very good at actually parallelizing programs. Interestingly, according to National Instruments a new feature of LabVIEW 2009 is parallel for loops, which can handle automatically multithreading loops [26]. It is unknown how capable the newest version of LabVIEW is at exploiting implicit parallelism, but it is notable that automatically parallelized for loops are at the core of the implicit parallelism exploited by the SISAL language, discussed below.

2.3.2 Mozart

Mozart is a VM and set of syntactic sugar for Oz.

2.3.2.1 Mozart Concurrency

The Mozart VM was build with concurrency in mind from the start. As a result it has good support for concurrent operations, one of the main attractions of using the language. Mozart threads claim to be very efficient and use fair preemptive scheduling. Unfortunately the concurrency is not implicit but explicit meaning threads have to be created manually. Threads are spawned by the following syntax:

```
thread S end
```

S is a function call or statement that will be executed on the newly spawned thread. A nice feature implemented by Oz is that threads will not execute on unbound variables. If the function spawned on a thread uses variables that have not been bound yet as input the thread will wait until all variables are bound before dispatching. This is similar to our idea of fine grain parallelism, described later in Chapter 4.

To test the concurrent performance of Mozart, a thread test was performed where 10000 threads were created; each yielded 10 times which causes about 100000 context switches. The test was performed in Mozart (Program 2.1) and Java 1.6.x (Program 2.2). The results of the test were very surprising. Mozart took only about 1-2 seconds to execute the test where as Java took about a minute and a half.

```
functor
import Application
define
   proc {Yield} {Thread.preempt {Thread.this}} end
   proc {Run N}
      {Yield}
      if N>1 then {Run N-1} end
   end
   Args = {Application.getCmdArgs
           record(threads(single type:int optional:false)
                  times(  single type:int optional:false))
}
   proc {Main N AllDone}
      if N==0 then AllDone=unit else RestDone in
         thread {Run Args.times} AllDone=RestDone end
         {Main N-1 RestDone}
      end
   end
   {Wait {Main Args.threads}}
   {Application.exit 0}
end
```

Program 2.1 – Example of Mozart Code.

```
import java.lang.*;
public class Death {
    public static void main(String[] argv) {
        int threads = Integer.parseInt(argv[0]);
        int times   = Integer.parseInt(argv[1]);
        for(int i=threads;i>0;i--) {
            MiniThread t = new MiniThread(i);
            t.start();
        }
    }

      private class MiniThread extends Thread {
          int n;
          MiniThread(int m) { n=m; }
          public void run() {
              do { yield(); n--; } while (n>0);
          }
      }

}
```

Program 2.2 – Example of Java code representing the same functional code as in Program 2.1.

2.3.2.2 *Implicit data Parallelism*

To test the implicit data parallelism in Mozart compared to C a simple program to compute the average of a large set of numbers was written in both C and Oz. Because of the difficulty of reading stdin in Oz the data set was hard coded into the application. Both programs used a data set of 2,000 integers between 1 and 99. Neither program was written to take advantage of parallelism beyond what their respective runtimes or compilers provided. Table 2.1 illustrates the average run time of each program over 100 runs.

Table 2.1 – Average runtime comparisons of C and Oz

	C – compiled via GCC	Oz – running in Mozart VM
Average run time over 100 runs	0.033 seconds	0.229 seconds

Even on a relatively small set of 2,000 numbers the C program performed nearly 10 times better than the Oz program. I believe this is due to the inherent speed gain in native compiled programs over the Mozart VM.

The other test that we felt would be good to run was to take standard deviation of a set of numbers. However after several hours of trying to implement standard deviation in Oz it was decided it was not worth proceeding. This illustrates the difficulty of programming things in Oz that are often very simple in other languages.

Based off what can be seen from the averaging there is no particular reason to use Oz for this project as it contains no implicit parallelism and without explicitly defining threads there is no speed boost gained over C.

2.3.3 SISAL

Based off of our initial impression of SISAL, we decided to learn more about it to evaluate its merits and potential usefulness. One thing that initially makes SISAL appealing is the fact that it "exploit[s] fully, implicit parallelism." [27] As one of our goals in this project is to exploit implicit parallelism, this means SISAL is at least useful as an example of how this has already been done and the potential performance gain involved. SISAL is made more appealing by the fact that it makes use of dataflow graphs in its compilation process [28]. The capacity to take code with no explicit parallelism in it and distill it into a dataflow graph, which would fully expose the opportunities for parallelism implicit in the program, is very powerful, and something we take advantage of to reach our goals. Also, in SISAL's time, i.e. the '80s-'90s, it was recognized as being as fast as if not faster than some of the most popular languages at the time, such as Fortran and C [29]. Another reason SISAL appears so potentially useful is the fact that the source code for SISAL's compiler is openly available in a SourceForge project and has been modernized, at least to the point of being ANSI C compatible, by one of its original contributors.

It was for these reasons, SISAL's capacity for extracting implicit parallelism and its availability to us as a framework off of which to base our own work, that we investigated SISAL more deeply. To verify claims about its performance, we compared the performance of a test program written in SISAL to a program performing the same calculations in C. The test program calculated the mean and standard deviation of all numbers from 1 to N. The results of benchmarking are shown in Table 2.2. The column

headers indicate the size of the series of numbers, N, used in the calculations, from 1 million to 300 million. The row headers indicate the compiler/language used as well as compiler options and/or number of threads used during execution. The results are in seconds.

Table 2.2 - SISAL vs. Optimized C Benchmarks for Mean/Standard Deviation Program

	1M	3M	30M	300M
gcc 4.1.2 (-O0)	0.034	0.094	0.944	8.999
gcc 4.1.2 (-O3)	0.025	0.068	0.732	7.183
SISAL (1 thread)	0.016	0.049	0.47	4.83
SISAL (2 thread)	0.02	0.025	0.25	2.62
SISAL (4 thread)	0.0165	0.014	0.14	1.41
SISAL (8 thread)	0.019	0.052	0.08	.83
SISAL (16 thread)	0.013	0.048	0.08	.79
SISAL-gss (8 thread)	-	.014	.09	.85
SISAL-gss (16 thread)		.013	.081	.79

SISAL's performance on this simple program, an example of the kind of calculations done in scientific computing, is impressive. It outperforms C, even with gcc's optimizations, in single-threaded performance, and with more threads the performance of course improves. However, while these are great results, they may not be typical. In this case, the core of the program we used consists of loops where each instance of the loop body is independent of the others. As a result, SISAL is able to parallelize these loops, which explains the consistent improvement in performance as the number of threads increases. This example gives us an idea of the kind of speedup we can only hope to achieve in the best case, with highly parallelizable programs. It should also be noted that SISAL appears to be capable of some other optimizations as well, given that it still solidly outperforms C with a single thread.

These promising results led to further investigation of SISAL. We examined the modernized SISAL compiler, called sisalc, made available by Pat Miller, who was a SISAL developer [17]. We also came across a package containing the original Optimizing SISAL Compiler, osc, including some useful documentation. From this we learned the compilation process SISAL uses to extract implicit parallelism from programs [30].

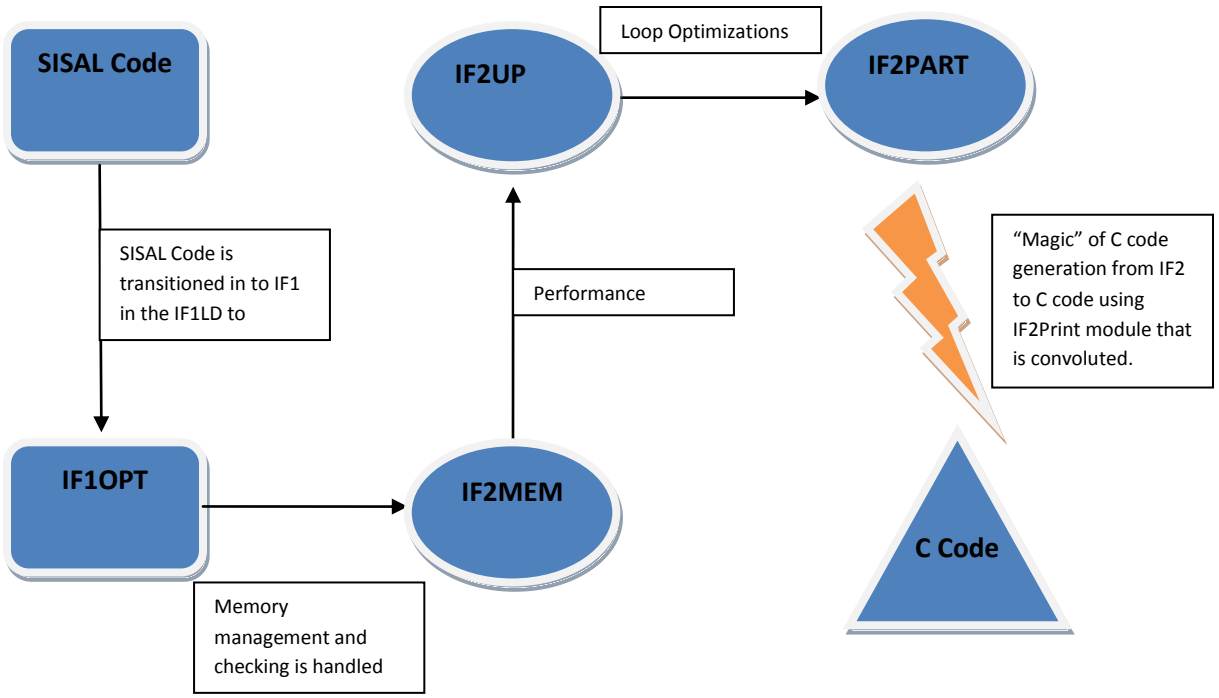

Figure 2.2 - Graphical representation of the SISAL compilation process.

The process goes as follows. The SISAL code from one or more files is parsed and transformed into the logical equivalent representation in Intermediary Form 1, or IF1. IF1 is a graph language that represents programs in terms of the flow of data between nodes of a graph. In IF1, nodes generally correspond to the basic operations performed by the program, such as an addition operation. There are some more complex types of nodes, called *compound nodes* in the terminology of IF1, to handle more complex language constructs such as branching and looping [31]. Some top-level optimizations are handled at this stage, such as invariant removal.

After optimization, the IF1 version of the program is transformed into a second intermediary form, IF2. We were unable to learn about IF2 as we did with IF1, but according to the compiler documentation the program undergoes most of its optimizations in the IF2 stage. These optimizations include optimizations to reduce memory transactions, loop fusion and unrolling, and, the core of SISAL's implicit parallelism capabilities, splitting loops into discrete chunks that can be executed in parallel. Once the IF2 version of the program has been fully optimized, it can then be processed and finally transformed in to C code. The code produced can then be compiled and will run using SISAL's own runtime environment.

Given that the compilation of SISAL takes place over the course of multiple, discrete stages, there appears to be plenty of opportunity to remake or rework one or more stages of the compiler in order to make use of what SISAL's creators have already achieved while reshaping the compilation process to meet one's own purposes. As a result, we made the decision to move forward with SISAL as a vehicle for the implementation of this project.

2.4 Related Work

As multi-core processors become more ubiquitous many large research divisions are focusing their attention on applying parallelism beyond the dataflow languages we've studied for this project. Intel's Thread Building Blocks [32] and Hadoop/Google's map/reduce [33] [34] are tailored to exploit parallelism available in large-scale computations, however require programmers to annotate or modify their programs to exploit parallelism. Sun Lab's Fortress [35] and parallelizing compilers like the Rice D HPF compiler [36] discover parallelism implicit in algorithms; however only address parallelism at the course-grained level. Our project's focus on implicit fine-grained parallelism with a graphical front-end is significantly different from the existing projects in the industry.

2.5 Summary

Parallelism, the execution of multiple operations simultaneously, is an issue that has in recent years become more prevalent due to trends in consumer commodity processor architecture. To adapt to this change in processor architecture, application designers must look beyond the problems addressed by the scientific community that we have presented in Section 2.1.1 in order to take advantage of parallelism in traditionally non-parallel application areas. To aid programmers in avoiding the hazards we have presented in Section 2.1.3, we investigated dataflow languages to discover examples of how to exploit implicit parallelism in programming languages. In this investigation we identified that the language SISAL shares many goals with our project, and due to these similarities, SISAL's modular architecture, and its use of a dataflow graph language, we decided to use SISAL as a base for the implementation of our project.

3 Design

The goal of this project is to create a graphical tool that represents a language that implements implicit parallelization and a runtime model that exploits it. After completing background research, we had to formulate a better in-depth design of our project as to meet our goal. This design requires a definition of how we intend to implement parallelism implicitly, how we intend to represent our language in the runtime, and a big picture architectural view of how the pieces of our project fit together to accomplish project goals.

3.1 Architectural Vision

The requirements for a *fine-grain parallelism* (FGP is discussed in the following sections) language are better met with a dataflow language than a systems language. After reviewing the languages outlined in our Background section we chose to use SISAL as a starting point for our project. SISAL was written in the 1980s to implement implicit parallelism. In order to utilize SISAL we had had to recognize the process in which it was compiled into useful form. This process is illustrated in Figure 2.2.

SISAL is more than just a dataflow language. It was a set of tools that converted a textual human readable dataflow language into a graph format. Part of the SISAL project also involved converting this graph format into runable C code. Because its project's goals seemed to align with ours, and the stages of the process seemed flexible, we used it as the base for our project.

It is important to recognize the distinct stages and to recognize where we designed our architecture to use existing technology. Using the existing technology provided by SISAL and a traditional C++ compiler we implemented a FGP language. The journey starts in the graphical front-end stage. This is the interface for our graphical programming language. Here a program is represented graphically as a series of nodes representing different actions. Our project converts this graphical representation into a textual representation in the language of SISAL. At this stage we use the existing SISAL toolchain to compile to a dataflow graph language known as IF1. Taking this IF1 file, we parse it into an in-memory format in a custom C++ code generation program. This converts this format into C++ code which utilizes our runtime library to simulate fine grain parallelization. This process is described graphically in Figure 3.1.

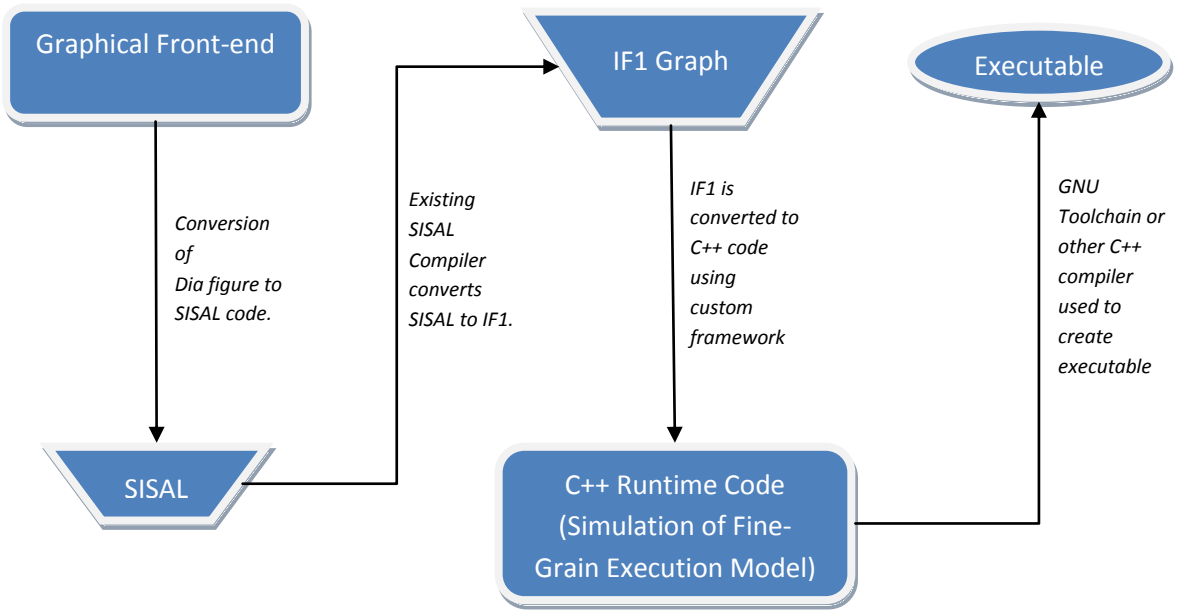

Figure 3.1 - Graphical overview of our design architecture

3.2 Fine-grain Parallelism

Conventional computer runtime models use stack-based execution for handling function calls. This model works excellent for abstracting through different layers of serial code. Computer architectures support function calls by saving registers which are pushed onto a stack when a function is called, and popped off and placed back into registers when the function returns. Functions are executed in a pre-determined order based on the order in which they are called by a program's code. Fine-grain parallelism uses a different idea for handling the concept of function calls. Fine-grain parallelism uses functions or blocks which execute off the heap when their inputs are ready.

Given the snippet of C code presented in Figure 3.2, we can conclude the sequence order of execution. Each line of code would be executed sequentially. This is because of the design of the C language and the nature of how it is translated into machine code. Each function call is set up to be called in order. When PreFunc() returns, it sets up the call to Func() and jumps to that code, when Func() returns the same process happens for Func2().

```
X = PreFunc(A,B)
D = Func(X)
E = Func2(X)
```

Figure 3.2 - A C Program

We designed a runtime using the notion of FGP. FGP builds on the idea of breaking a program into logical blocks each with inputs and outputs connecting to other blocks, where logical block are functions or fragments of functions. Blocks are able to execute when all of their inputs are ready. When a block finishes, it publishes its output notifying dependent blocks that information is available. Parallelism is exploited if the number of ready blocks at any given time is greater than one. Using the C example from above, this same code could execute utilizing the concept of FGP in an equivalent manner. This is represented graphically in Figure 3.3. We can see that because both Func() and Func2() require the same input, they can both be run concurrently. This only is valid if we are guaranteed the execution of Func() and Func2() do not interfere with each other. This guarantee is present in a dataflow language because there are no shared variables.

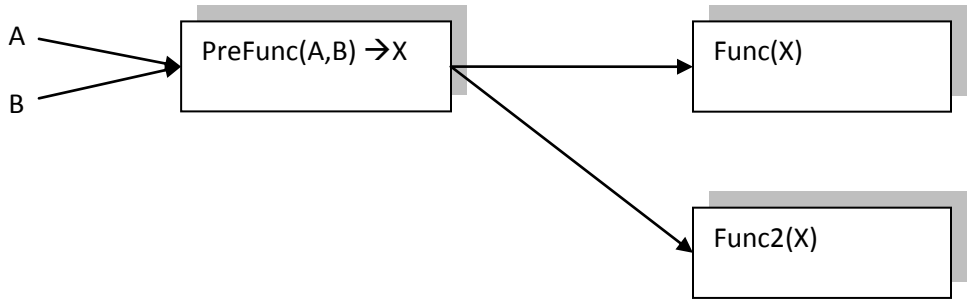

Figure 3.3 - A graphical overview of the power of fine-grain parallelism. Because Func() and Func2() both require X, both can be executed at the same time.

In a language like C, this only works under the best of circumstances. Because C is a systems programming language it allows the programmer to have near complete control of execution. If Func or Func2 write to S, a shared variable, at the same time then the transformed execution model is not the same computation. If the two functions write to any global or shared data area, then the execution is not equivalent. Ultimately, C is a poor choice of a language for a fine-grained runtime model because it has such loose rules for accessing data. While it is possible to implement fine-grained parallelism in C, there is no contract for calling functions to guarantee that abstraction is safe. For FGP, blocks must not write to global memory and only alter global memory through their outputs. This problem opens the door for how to represent a fine-grained parallelization language.

By using FGP as our implementation vehicle for implicit parallelization, we will move beyond the loop based solutions employed by other implicit programming languages. Breaking up a serial computation into a series of potentially parallel execution blocks is a process that will also take us a step further than OpenMP or Thread Building Blocks loop based explicit methods.

3.3 Representation of Fine-grained Parallelism

As we have described previously, a node can fire whenever its inputs are ready and can run to completion to produce its outputs, and the execution of a node is effectively non-interruptible. This differs strongly from a traditional stack-based approach. In a more direct translation of a stack-based approach one function can fire and then yield to call another function waiting on that function's

outputs. This works well in a stack-based approach, but causes issues with fine grain parallelism both in representation and implementation.

To compare a traditional stack-based execution model to a non-interruptible execution model, first consider a function f that calls two functions g and h as shown in Figure 3.4. The arrows indicate the flow of data, meaning the function f receives some input that is passed on to functions g and h, and the outputs of g and h become the outputs of f. In the traditional model, the execution of function f will be preempted in order to execute g and h. The function can be restructured as in Figure 3.5 to avoid preemption during execution of its activation records. Note that if g and h in turn call other functions, their activation records may be expanded similarly, just as the activation record of f was here, but this graph illustrates the general intent.

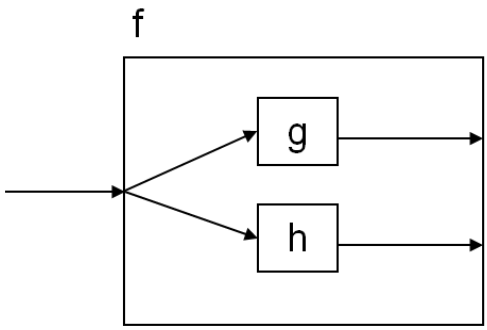

Figure 3.4 – Dataflow graph of simple function call

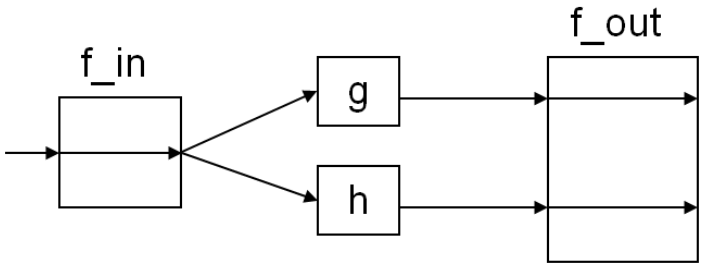

Figure 3.5 – Graph of non-preemptable function call equivalent

This rearrangement affects the frequency of context switching and the transfer of data and control between activation records. For example, take the more realistic problem of calculating standard deviation and arithmetic mean of the set of numbers from 1 to n. Figure 3.6 contains the dataflow graph based on a SISAL implementation of the program.

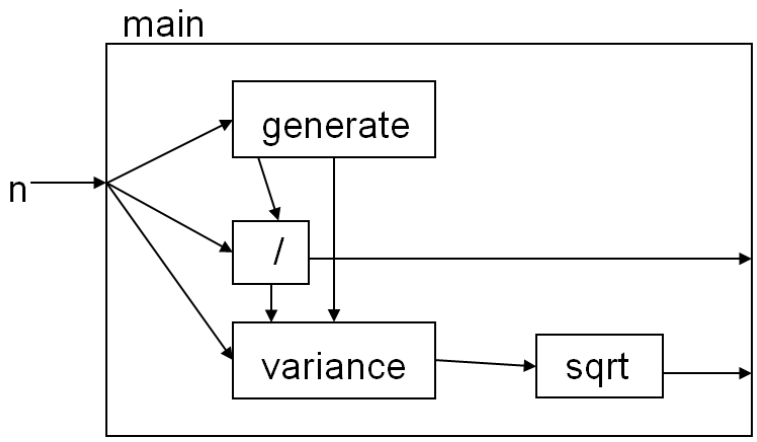

Figure 3.6 – Dataflow graph of SISAL mean and standard deviation program

Assuming division and square root are built-in functions that can be performed within an activation record's execution without a context switch, and not counting any switching within calls to generate and variance, this program will have a total of six context switches with preemptable activation records, when:

1. main begins
2. main calls generate
3. generate returns to main
4. main calls variance
5. variance returns to main
6. main returns

Now take the same program redrawn in Figure 3.7 for the use of non-preemptable activation records.

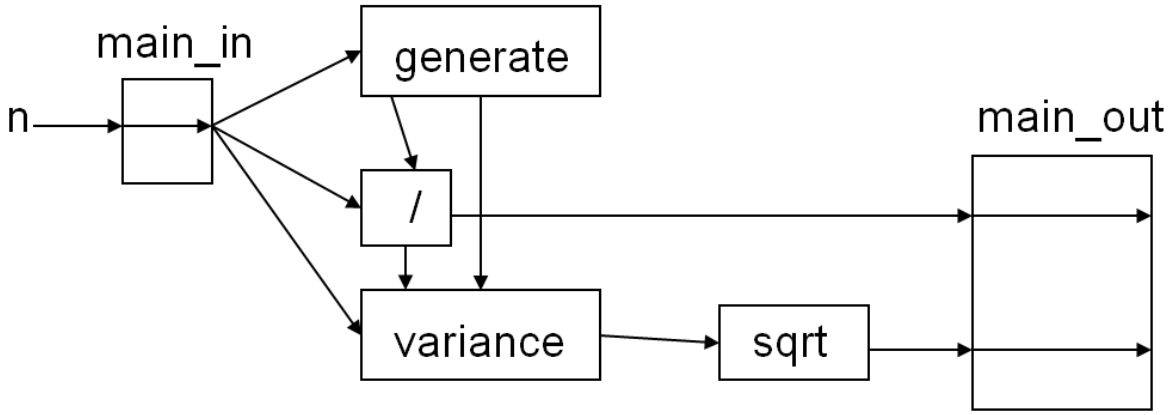

Figure 3.7 – Dataflow graph of non-preemptable mean and standard deviation program equivalent

The program, modeled in this way, will have five context switches:

1. main_in begins

2. main_in passes off to generate
3. generate passes off to variance
4. variance passes off to main_out
5. main_out passes off to some consumer of its output

Even in a program that is not parallelizable at the top level, changing the activation record model in this way should at least make it no more expensive in terms of context switching than in the original model. Furthermore, if we are able to eliminate "in" and "out" activation records through graph manipulation, we can reduce context switching even more. Even if we cannot, in functions that call many other functions the cost of the extra in/out context switches will be masked by the savings on the cost of the internal context switches. In functions that are more parallelizable, we should be able to further reduce context switches by passing off control/data to multiple functions simultaneously.

Finally, the impact of this model on the ability to exploit parallelism is strong. In the first example, function f seemingly calls g and h in a parallelizable manner. Using preemptable activation records, we will have an activation record which contains a function pointer to some function representing f, which then calls g and h. Assuming there is a way for the activation record to be preempted and transfer control to another function's activation record, then since the activation records themselves would be implemented in a procedural language, a problem presents itself: How can it call both functions simultaneously in order to take advantage of the apparent parallelism? This problem is avoided by using our non-preemptable activation record model. In order to distinguish the equivalent of activation records in this non-interruptible model from the activation records of the traditional stack-based model, we decided to call our implementation of activation records *activation atoms*. This is because their purpose for us is less that of a record for a preempted function call than the basic, indivisible, atomic unit of execution in our model.

In addition, using non-preemptable activation records appears to pose no problem to representing basic control structures such as loops and if-then-else constructs. Pat Miller discussed this somewhat in a talk he gave on SISAL and parallelism. He was a SISAL developer and is responsible for the SISAL project on SourceForge [17]. We contacted him as a part of our investigation of SISAL, and he came to give a talk at WPI as a result. A for-loop, where no data is passed between iterations of the loop, can be decomposed as in Figure 3.8.

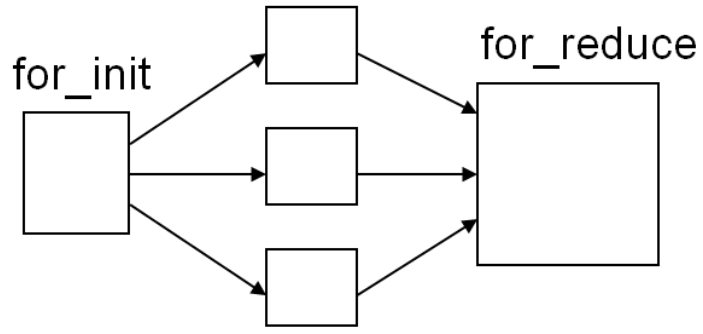

Figure 3.8 - Non-preemptable for loop equivalent

For_init provides the necessary data, such as loop counters, to the middle activation records representing loop iterations, and the results are combined and passed on by for_reduce. If-then-else constructs could be broken down as in Figure 3.9.

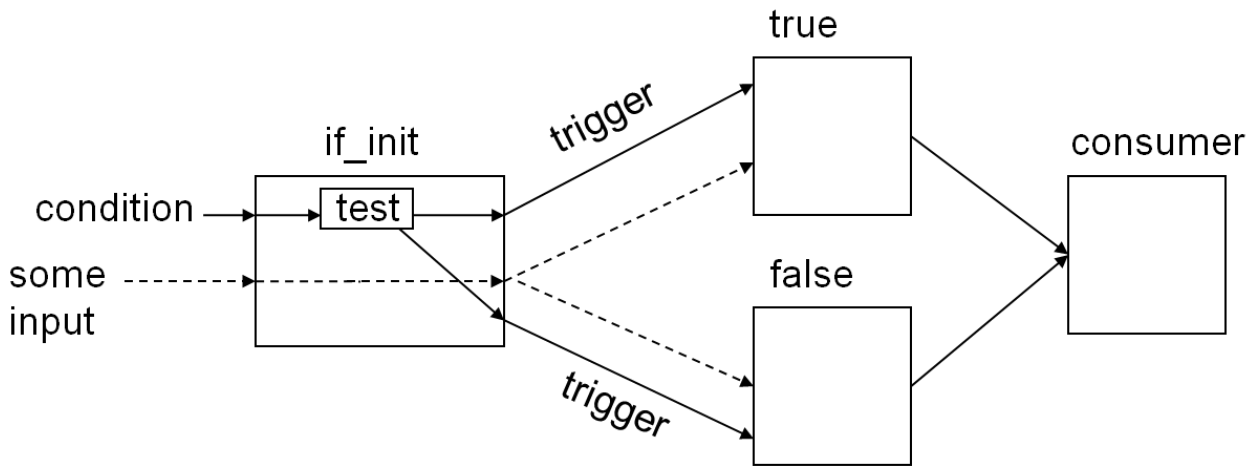

Figure 3.9 Non-preemptable if-then-else statement equivalent

However, it would not be necessary to break all operations down into multiple activation records. Non-parallelizable portions of code, like loops with dataflow between iterations, could potentially be put into a single activation record to avoid unnecessary context switching. If-then-else constructs could also potentially be kept whole.

In conclusion, while in traditional, stack-based execution models it makes sense to have an activation record for each function call and to allow activation records to be preempted, it makes much less sense when trying to maximize fine-grain parallelism using a heap of activation records.

3.4 Summary

In light of these design factors and architectural vision, our project has three major components. The first is the graphical modeling of a program and the subsequent transformation into SISAL code. The graphical front-end and SISAL code allow us to have both a graphical and textual interface to the dataflow representation of IF1. The second piece of the project is a C++ library that abstracts the ideas of fine grain parallelism such that we can prototype different applications using a general-purpose fine-grained parallelism runtime library. This library handles thread and memory management for the purposes of fine-grained parallelism. The last architectural piece is the conversion of IF1 into C++ code that uses our FGP runtime.

4 Graphical Front-End

4.1 Introduction

As part of our study of improvements in parallel programming we investigated the feasibility of a graphical programming tool to ease the development and recognition of parallel constructs within traditionally sequential programs. To this end we designed and implemented a graphical programming tool within the open source drawing tool Dia.

4.1.1 Graphical Programming's Appeal

The primary appeal of a graphical programming environment its ease of use. Graphical IDEs available on the market today prevent users from creating syntax or semantic errors by not allowing them to connect components, which don't fit together. Additionally limiting input from unconstrained text to a relatively small set of graphical symbols reduces the learning time of a language. In our case the massive amount of parallelism that can be exploited by our language make it a good fit for scientific applications. In many cases people creating code in these environments will not be as experienced programmers.

For our purposes programming a parallel program in a graphical language is extra appealing. By not being constrained to the logical line-by-line flow that a text file introduces it becomes easier to recognize where sections of code can be run in parallel. We believe this will help people visualize potential parallelism in their applications and use that knowledge to tweak their programs layout increasing available parallelism.

4.1.2 Dia

Dia is an open source graphical modeling toolkit. It was selected because it provides an easy way of visually creating directed graphs, and it was easily extensible through XML and Python. Using the extension functionality would allow us to graphically design programs in Dia, save their graphical representations to Dia's default format and then export them to SISAL files.

Initially we considered Graphviz [37] for the creation of our graphical front-end. Although Graphviz has a large community following we found that the application is mostly focused on displaying graphs and no standard graph editor existed. We decided that it would be easier to implement our front-end in Dia and therefore moved forward in that direction.

4.2 Design

The design subsection focuses on two primary design concerns from our font-end, the design of Dia shapes and the design of our function creation mechanism.

4.2.1 Shape Design

Dia's main mechanism of extension is through shapes, objects that Dia uses as nodes in its visual graphs. A shape is defined either within a C dynamic library or an XML file. Although more powerful, the C API for Dia is poorly documented, because of this we elected to create shapes solely using XML files.

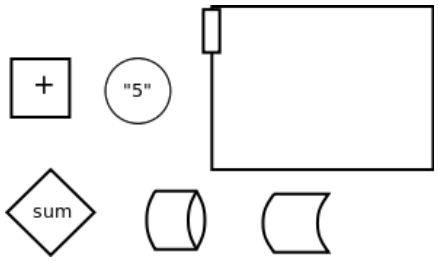

Figure 4.1: Clockwise from top left: basic operations, constants, loops, inputs, outputs, and array operations.

In designing our shapes we attempted to maintain a consistent language across all our shapes. We defined six general types of shapes: Basic operations, Constants, Loops, Input, Output, and Array operations. Figure 4.1 demonstrates the overall shape given to each of these types allowing them to be visually distinguished from each other. Additionally in development we were able to use our first constructed node of each type as a template for future nodes decreasing development time.

4.2.2 Loops

Loops in particular deserve special attention. While the remainder of the shapes we have implemented can be viewed as "black boxes" which transform data from their input to their output we require designers to layout the code executed within a loop.

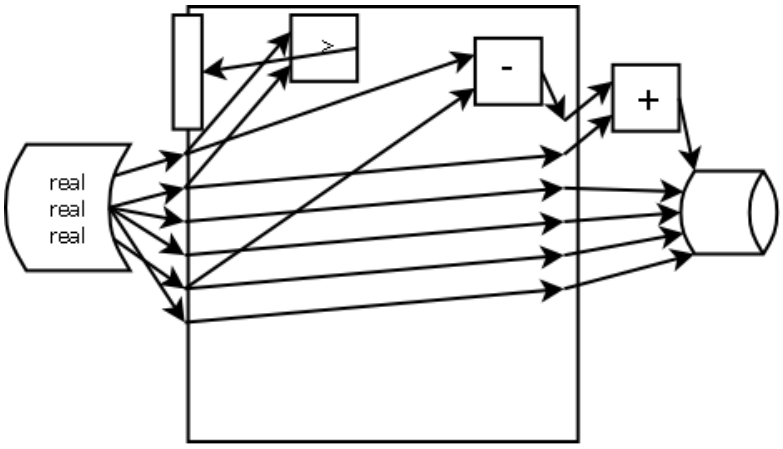

Figure 4.2 – A decrementor in our front-end

Dia provides a mechanism to detect collision between two shapes, we are able to use this to create a visual space in which a designer can layout the inside of a loop. Figure 4.2 shows a loop implementing a simple decrementor.

4.2.3 Designing Functions

When programming in a textual language functions are a useful tool, both for allowing code reuse and creating divisions in the code. Graphically functions should serve the same goals. To achieve this we needed to design a way of dealing with functions, which would allow us to:

- Define a function in a visually isolated space.
- Reference that function, possibly multiple times using our standard language of visual node connections.

Our approach to functions was to require each function be defined within a separate file as in LabVIEW. This provided use with three advantages:

- Removed the need to figure out which nodes belong to which function when exporting code.
- Each function is visually isolated from the rest of the code.
- Dia supports the ability to have multiple files open at once but not the ability to define multiple visual contexts within an open file.

To implement a visual function call mechanism we simply needed to generate and install a shape file for each function that was created, afterwards the shape representing the function would appear in the Dia palette.

4.3 Implementation

Dia supports extension through three means, python plugins, C plugins, and XML files. Although the C plugin system is the most powerful is also sparsely documented, in contrast Dia provides tools to generate the XML files and the python plugins can take advantage of introspection methods available to that language to discern the structure of the API without documentation. For our purposes we have focused on the creation of XML shape files and a python plugin to export our graph to SISAL source files.

4.3.1 Initial Implementation

The original Dia to SISAL front-end was designed to iterate through a series of Dia shapes generating SISAL code as it progressed. This initial iteration of the translation program was basic; there were only two possible types: Integers and Booleans, and only arithmetic and Boolean operations were supported.

4.3.1.1 Typing

Our initial approach to specifying the types of variables was simple. SISAL uses type inference, so types only need to be declared in the parameters of a function. In our implementation function's types were user specified in the text of the starting shape. Operations were mapped directly to the types they were meant to output, this ignored the fact that arithmetic operations output a type dependent on their input. At this stage this flaw was ignored in favor of getting something that worked to build off of.

4.3.1.2 Translation Function

To get the body of SISAL code from the Dia program, the translation function iterated through all the shapes to till it found the starting shape. The export function would then be run on each of the lines emanating from the starting shape. This export function took as a parameter both the shape to export and a list of the shapes already exported. It would check that all of the shapes, which this one depended on, had already been exported. Only if this were true the export function would generate SISAL code for this shape and append the shape to the list of exported shapes. This system of selectively

avoiding handling shapes until all previous shapes were handled was done to ensure that the exported code would not try to use any values that had not been created yet.

4.3.1.3 Naming

There was some experimentation with ways to generate variable names, such as using the object id or having them be assigned by the program, it was decided that using the coordinates of the shape the variable was related to would be the best idea. This had the advantage of being a unique identifier for each shape and could be derived from the shape itself.

4.3.1.4 Debugging

Python uses an exception system, however Dia catches exceptions that Python plug-ins generate and does not display error messages. This made the plug-in take significantly longer to debug since all that was often reported was that an error had occurred. We eventually found a way to make this information available. In retrospect it probably would have been a good thing to focus on getting done before anything else.

4.3.1.5 Initial Results

The result of our initial work was a plug-in that could produce SISAL code capable of performing basic mathematical operations on a given set of numbers. Figure 4.3 illustrates the graph for the basic operation (a+b)*c.

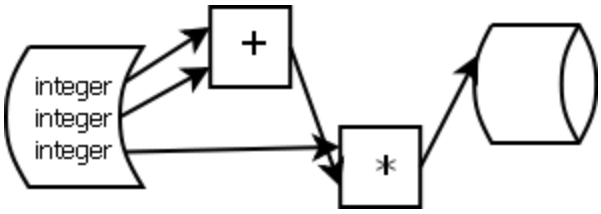

Figure 4.3 - f(a,b,c) = (a+b) * c In the graphical front-end

```
% SISAL source code
define main
function main(n536a1102:integer;n524a1015:integer;n536a927:integer;returns
integer)
let aeoirsh := 5
m1380a785 := n536a927 + n524a1015
m1820a1165 := n536a1102 * m1380a785
in m1820a1165
end function % main
```

Program 4.1 - f(a,b,c) = (a+b)*c In SISAL

The Dia graph presented in Figure 4.3 generates the SISAL code in Program 4.1. The parameter line is determined by the starting shape, and the last line of the function comes from the multiply shape. In this example each shape translates into a SISAL variable.

4.3.2 Revisions

Based on the work we did getting the initial implementation of our front-end functional we identified several areas where our implementation needed revisions. We addressed these in a subsequent revision of our front end.

4.3.2.1 Variable Naming

In order to support for loops and other SISAL constructs, we needed a single operation to return multiple values. This was not possible with our existing system as variable names were created with shape coordinates, of which there is only one for each shape. Instead the variable names were generated from the output nodes of each shape. This allowed a single shape to generate multiple output values.

4.3.2.2 Typing

Our type system needed revision too, in order to support operations like the basic arithmetic ones, whose output types are determined by their input types. This was achieved by writing a function to calculate the output types based on the input types and the operation being performed. Those output types were mapped to node names, and this map was passed along to every subsequent translation of an operation. Initially, this code was only implemented with variables of type real, when it was shown that the variable changes were working subsequent types were added.

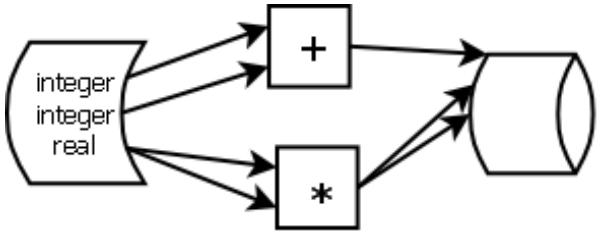

Figure 4.4 -: f(a,b,c) = {a+b,c^2,c^2} In the graphical front-end.

```
% SISAL source code
define main
function
main(n77696875778:real;n76456905:integer;n77696875603:integer;returns
integer,real,real)
let aeoirsh := 5
in let m176559855 := n77696875778 * n77696875778

in let m176559855 := n77696875778 * n77696875778

in let m168554955 := n77696875603 + n76456905
in m168554955,m176559855,m176559855
end function % main
```

Program 4.2: f(a,b,c) = {a+b,c2,c2} In SISAL.

The Dia graph in Figure 4.4 and the SISAL code in Program 4.2 represent an example of the type inference. The types of the arguments for main are specified in the input shape. The return types are calculated based on the return types of the operations.

4.3.3 Implementing Loops

Loops introduced several difficulties apart from the revisions to the typing and node system. SISAL code for existing shapes was generated based on a template with spots for the input and out variables. Loops would require more work as the code within the loop needed to be generated as well as the loop boundaries.

Loops are primarily differentiated from other shapes in that they behave like the program as a whole with constituent shapes. Unlike the function though a loop shape acts as both the starting and ending point of the loop's body. We were able to exploit the similarities between functions and loops by using the same algorithm to generate the output code for both. This implementation was complicated by a possible case where a loop shape could be connected to itself; this caused the program to enter an infinite loop, but was easily handled by treating this as a special case and inserting code to detect and handle it.

```
% SISAL source code
define main
function
fun0(m1674a1570:real;m1674a1671:real;m1674a1772:real;m1674a1873:real;m1674a1974:real;m1674a2075:r
eal returns boolean)
let a5junk76536893:=4
 m2750a1320 := m1674a1570 - m1674a1974
 m2195a1250 := m1674a1570 > m1674a1671
in m2195a1250
end function % fun0
function main(n1461a1825:real;n1447a1730:real;n1461a1635:real;returns  real,real,real,real,real)
let aeoirsh := 5
 m2818a1468,m2818a1570,m2818a1671,m2818a1772,m2818a1873,m2818a1974:=for initial
m1674a1570:=n1461a1635
m1674a1671:=n1447a1730
m1674a1772:=n1447a1730
m1674a1873:=n1447a1730
m1674a1974:=n1461a1825
m1674a2075:=n1447a1730
while fun0(m1674a1570,m1674a1671,m1674a1772,m1674a1873,m1674a1974,m1674a2075) repeat
 m2750a1320 := old m1674a1570 - old m1674a1974
 m2195a1250 := old m1674a1570 > old m1674a1671
m1674a1570:= m2750a1320
m1674a1671:= old m1674a1671
m1674a1772:= old m1674a1772
m1674a1873:= old m1674a1873
m1674a1974:= old m1674a1974
m1674a2075:= old m1674a2075
returns value of m1674a1570
value of m1674a1671
value of m1674a1772
value of m1674a1873
value of m1674a1974
value of m1674a2075
end for
m3164a1400 := m2818a1468 + m2818a1570
in m3164a1400,m2818a1671,m2818a1772,m2818a1873,m2818a1974
end function % main
```

Program 4.3 - f(a,bc) = while(b>a){b-c}; b + a In SISAL

While loops deserve special attention as they introduce the concept of a recirculant value. Recirculant values are used to pass information between iterations of the while loop. This can be seen in the Dia graph in Figure 4.2. The results of a greater-than operation are tied into the conditional for the loop, and the values exiting the loop on the right hand side correspond to the values entering the loop on the right hand side. Determining the value of the conditional is done by running the shapes in the

loop through the main function, once initially before generating the main body of the loop, stopping when it reaches the conditional node. This generates the code representing the series of operations composing the value of the conditional. Unlike for loops, while loops don't use the type values from their internal operations for output types, instead using the type values corresponding to the inputs. Program 4.3 shows the SISAL code that is generated from a while loop like this. Due to limitations imposed on SISAL with respect to the depth of let-in statements the conditional check for the loop has been moved to a separate helper function.

4.3.4 Implementing Constants

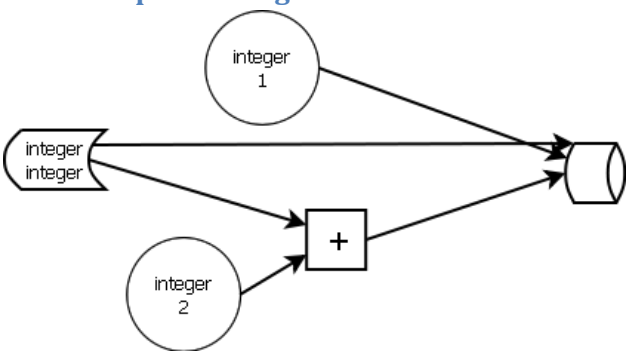

Figure 4.5 - f(a,b) = {a,1,b+2} In the graphical front-end

Constants are also a required part of what was needed for the creation of more complex programs. Figure 4.5 demonstrates the use of constants in the graphical front-end. The process to translate them into SISAL code consisted of iterating through all the shapes on the screen finding the constants, and saving the type and value. The constants can then be assigned to values at the beginning of the main function.

4.4 Known Design and Implementation Concerns

Our decision to use Dia and our design of functions introduce four primary design and implementation flaws that cannot be solved while we use Dia.

- **Visual Clutter** - Dia provides no mechanism for laying out node connections to minimize over laps.
- **SISAL file Clutter** - Each SISAL file generated can have at most one function defined. Even simple programs will require an unnecessarily large number of files.
- **Palette Clutter** - Each function that is created will introduce a new shape in Dia's tool palette. This will quickly be unmanageable even for small programs.
- **Generic Usage** - Dia does not differentiate between a graph representing data flow information and one representing a UML schema. We need to make sure we have the former before attempting to export SISAL code.

These issues do not hamper our use of Dia as a proof of concept for the graphical programming language.

4.5 Evaluation

4.5.1 Installation

In order to use our graphical front-end you must first obtain Dia. It is hosted courtesy of the Gnome project at http://live.gnome.org/Dia. Dia is available for windows (via a link on their website) or Linux via many distribution's repositories. It can also be built from the source available on their website. To install our export plug-in the Python extensions must be enabled. There is an option at build time to disable these. We have encountered versions built for Fedora 11 with the Python extensions disabled; the easiest way to build without them is to obtain the source from the Fedora repository's CVS and modify the spec file to build an RPM with the Python extensions enabled.

After obtaining a working copy of Dia with the Python extensions installation of our plug-ins is as simple as adding our export plug-in and shape files to the plug-ins directory.

4.5.2 Using the Interface

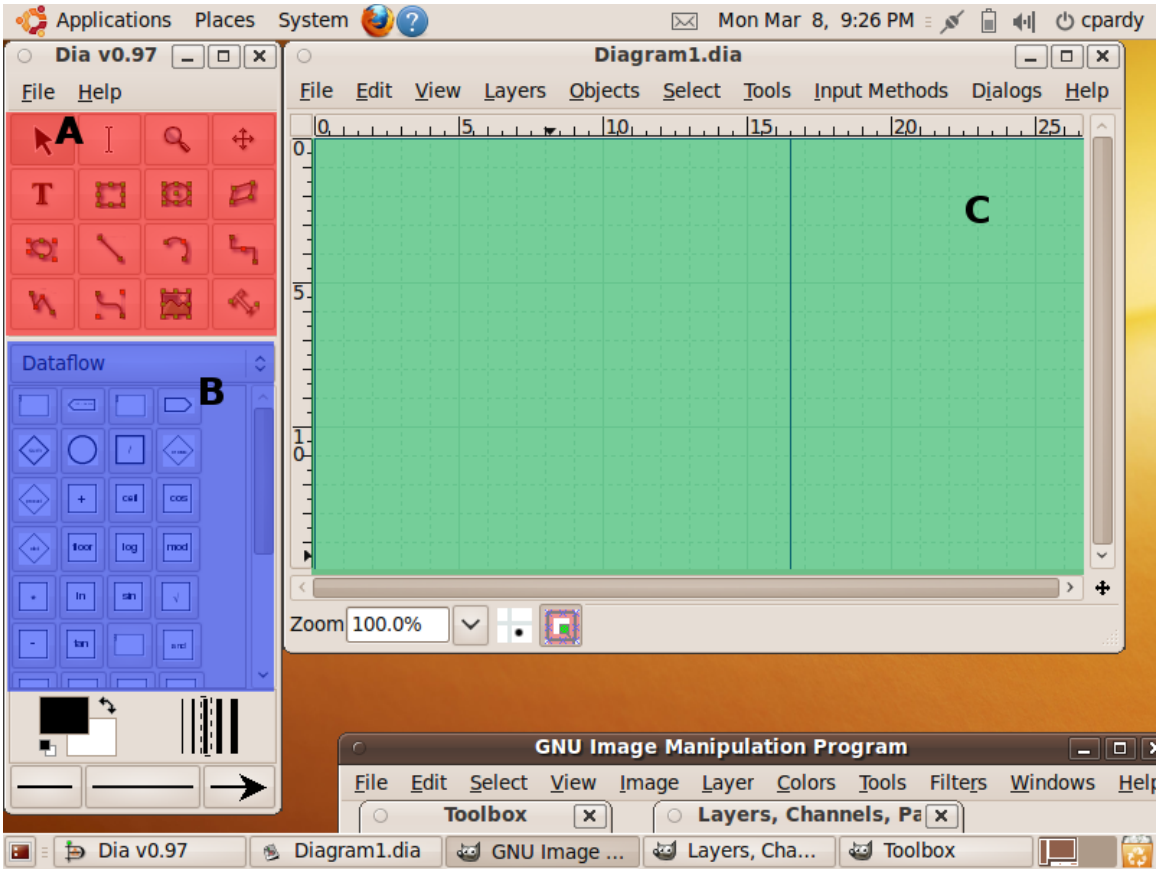

Figure 4.6 - The Dia interface in Linux

The Dia interface is primarily made up of three parts labeled in Figure 4.6 as A, B, and C.

A. The tool pallet. This provides generic Dia tools such as lines, bezier curves, text and select/move tools.

B. The shape pallet. This provides the shapes for a given sheet (collection of shapes). We currently have the Dataflow sheet selected.

C. The drawing canvas. This is the area where shapes are laid out and connected. It includes options to snap shapes and lines to the grid and zoom the view.

To add shapes from the shape pallet to the canvas simply drag and drop the icon of the shape onto the canvas. Alternatively clicking the icon for a shape will select it. After a shape is selected clicking on the canvas will place the shape at your cursor.

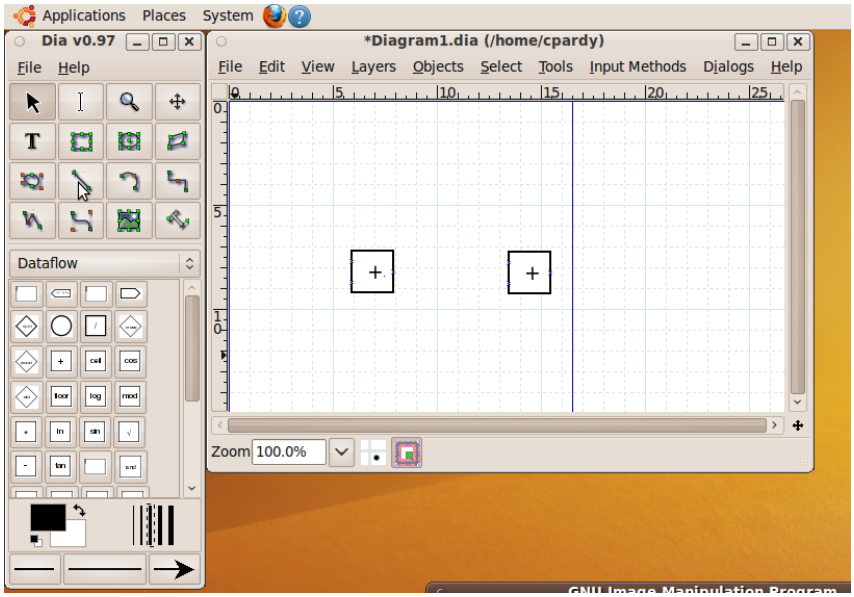

Figure 4.7 - selecting the line tool

After multiple shapes are added to the canvas the line tool can be selected from the canvas as in Figure 4.7. The Line tool is used to connect the outputs of one shape to the inputs another.

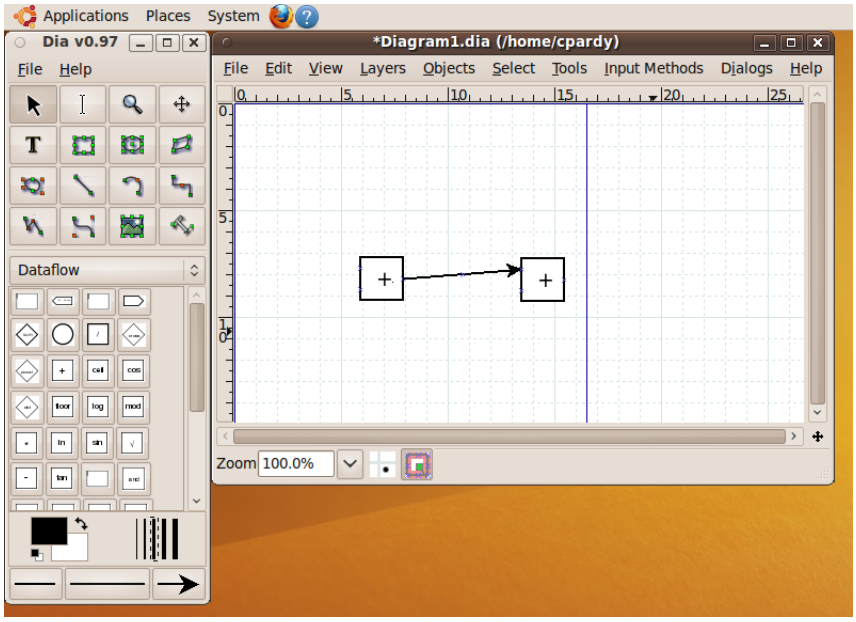

Figure 4.8 - Drawing a line

To draw a line, simply select the line tool and then drag from one of the output nodes of one shape to one of the input nodes of another. This will create a line connecting the two shapes as in Figure 4.8. After you understand the method of creating shapes and lines you have all the knowledge you need to create Dataflow graphs in Dia. By simply expanding the shapes we use we can create richer programs.

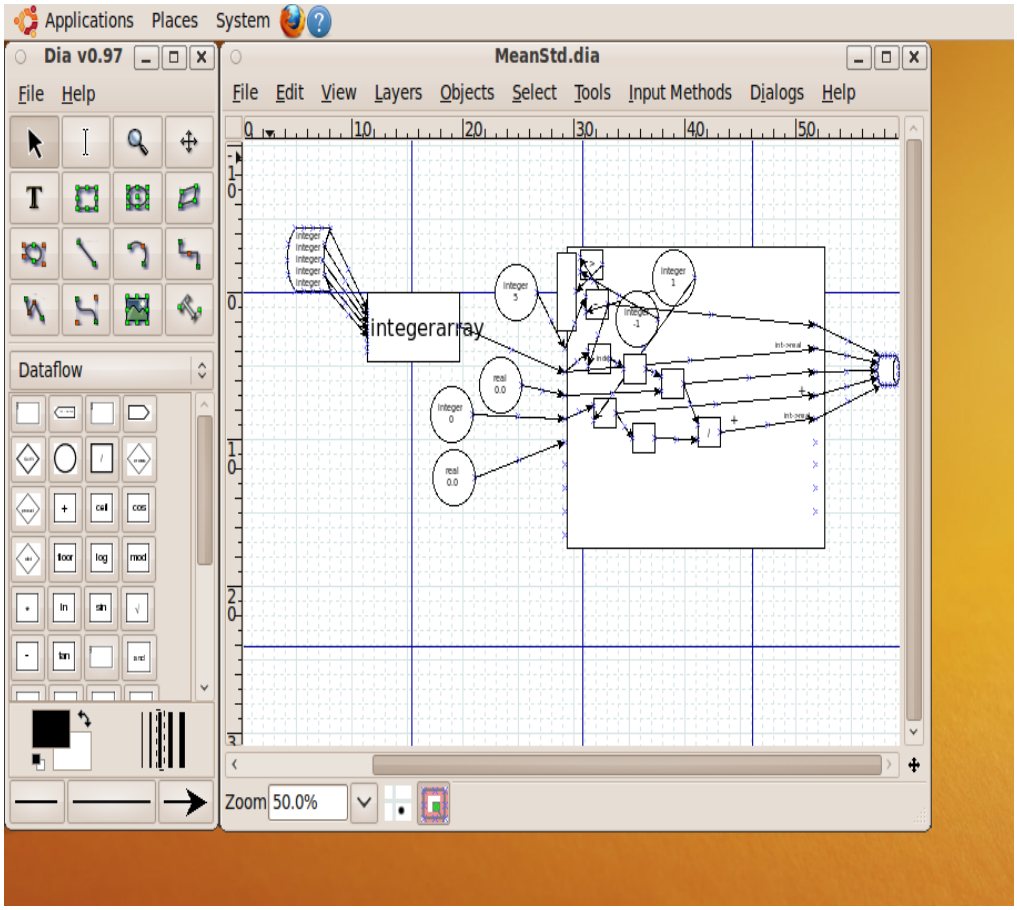

Figure 4.9 – Mean and Standard Deviation laid out in Dia.

Figure 4.9 shows a program that calculates the mean and standard deviation of its inputs build inside Dia. It makes use of the loops, constants, basic operations, inputs, and outputs discussed in the design section of this chapter.

After a suitable program has been created the Dia graph should be exported using our Python export plug-in. To do this go to File → Export… and select "SISAL source code (.sis)" from the drop down list of export types. Click export to generate SISAL source code from the Dia graph. Figure 4.10 shows the export dialog.

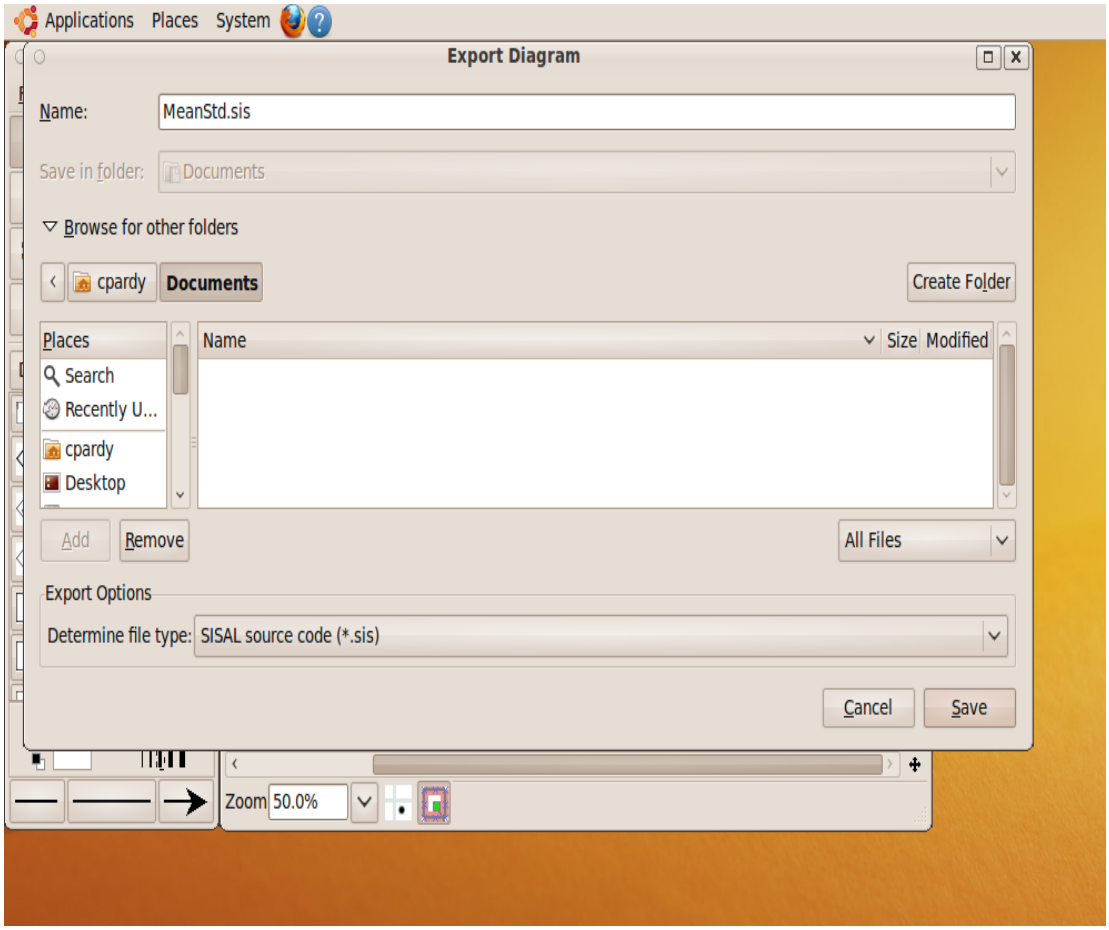

Figure 4.10 – Exporting the MeanStd diagram to SISAL.

4.6 Summary

We have designed and implemented a graphical programming front-end using the open source graphical modeling program Dia. Our implementation, although intended as a proof-of-concept is feature complete and supports the creation of for and while loops, constants, and the full set of basic operations. Our design has focused on ease of use and used a consistent vocabulary of shapes to represent each of, basic operations, loops, array operations, input, output, and constants. Our implementation of an export plug-in has been refined through successive iterations to support all of the features of SISAL, overcoming numerous challenges along the way.

5 Runtime Implementation

While both threading and memory management is discussed architecturally beforehand, numerous implementation level details thwarted a one-to-one mapping of the initial design to implementation. The architectural design is best described as a detailed sketch, where as the implementation design was a detailed blueprint that took into consideration system limitations. This chapter focuses on the implementation of our prototype of fine-grain parallelism. Included is a discussion of the design decisions we made related to thread and memory management. This chapter begins with an in-depth overview of our threading and memory design, transitions into a state representation of the runtime, and finishes with a discussion on a code-level example.

The Design chapter focused on the idea of fine-grain parallelism, where the abstract idea of a block of execution is discussed. These blocks execute when their inputs are available. This execution model is implemented under the name of ActivationAtom. When discussing ActivationAtoms we are referring to the implementation-level detail of a fine-grain execution block. Therefore we define an ActivationAtom as a programming abstraction for a finite unit of parallel programming code. Our programming abstraction provides all the functionality required for the usable deployment of fine-grain parallelism as described in Chapter 3. We also use the non-interruptible model that was presented in Chapter 3.

Our runtime was implemented to prototype the feasibility of a fully fledged fine-grain parallelism runtime. It was not implemented to be optimized for performance, or optimized for any particular hardware or software platform. The runtime was written in C++ using cross-platform programming techniques that would allow it to execute on many different systems. While writing the runtime at this high level sacrificed performance, it allowed us to develop and debug the runtime faster.

5.1.1 Threading

To handle multiple ActivationAtoms executing at the same time, a utility needed to be used to share CPU time. The obvious choices were processes or threads which both allow a time-division multiplexing of CPU resources. Processes were deemed too heavy-weight to meet the design standards, thus threads were used. To further minimize the overhead of threading, we decided to map ActivationAtoms to a thread pool.

A thread pool is a set of inactive threads which belong to a process. Each thread sleeps until it is woken up upon a signal notifying the potential for work. The thread pops the FIFO queue representing tasks and executes the stalest task. Upon completing the task, the thread looks for more work, if none is available it returns to sleep. This is represented graphically in Figure 5.1.

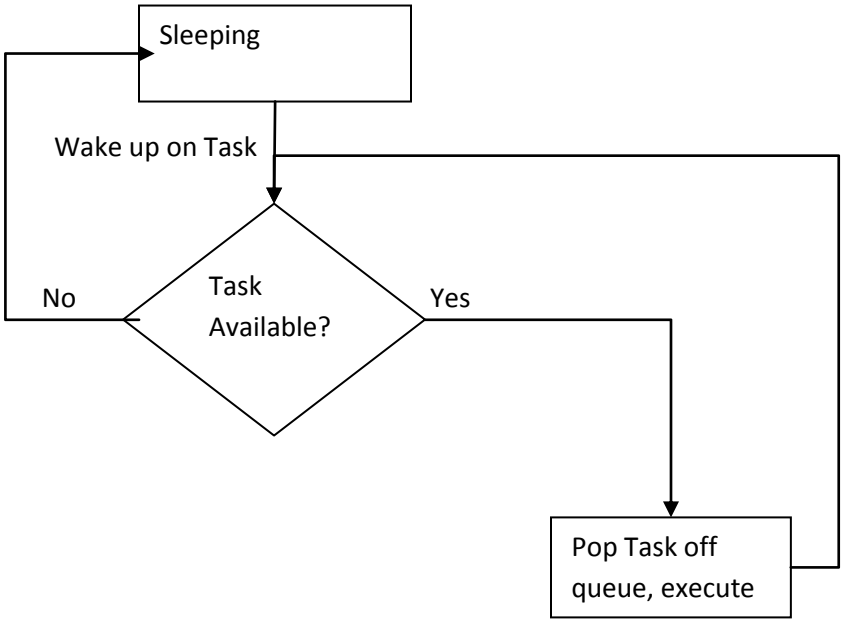

Figure 5.1 - Thread Pool Model

A thread pool is a good model for our project because it promotes low-latency for starting tasks. Spawning off a new thread for each task would create operating system overhead for each task. Because we utilized fine-grain parallelism, which could result in hundreds of tasks per process, this dispatch overhead would be measurable.

In this model, ActivationAtoms are then treated as fibers which are executed on one of many threads. There can be many fibers waiting in the task queue before it gets a chance to execute on a thread. Implementing our non-interruptible block ideas as described in Chapter 3, each fiber executes to completion without being yielded to another.

5.1.2 Memory

Our implemented memory management system needed to accomplish both thread-safe access to output/input regions of an ActivationAtom and manage memory properly for stale or finished ActivationAtoms. The system needed to support the ability to safely access the inputs and outputs of ActivationAtoms in a thread safe manner with an arbitrary number of threads. The best solutions involve minimizing the amount of locking to synchronize the system. Relying on locking to synchronize memory would increase the amount of code that is not parallel. Because our project is aimed at having a maximum amount of parallelism, atomic operations are a better fit.

While an input-centric view is what current function call mechanisms use, it does not fit well for a fine-grain parallelism implementation. Upon examining the issue of synchronizing memory access, we

instead chose to take an output-centric view of the memory organization. This choice was born out of the powerful concept that an input's location can be known earlier than its value; By allocating space for the value before it is actually published, all entities can know where the value will be published before the data is valid. Specifically, data is stored as an output and pointed to as an input. Because many ActivationAtoms can finish at the same time, this avoids the clash of publishing their outputs all to one memory area. If the inputs were directly written then there could be many threads accessing the same memory region at the same time for a write operation. By orienting the structure in the output direction, only one thread writes to a region of memory. This avoids any potential data collisions.

Essentially, our implementation gives each ActivationAtom its own storage space to store its output. This storage space is visualized as a slab in the processes heap. The slab contains all the information of the block, specifically the activation record. Other threads only read from this output space once the original block is finished. In Figure 5.2 we see a graphical representation of the ActivationAtom in memory. There is space to store the outputs directly, as well as a count of how many uses are left for all outputs.

Figure 5.2 - Graphical overview of a slab in the slab allocator. The entire slab is the size of a cache-line and stores the outputs for a block. Future blocks can only access these outputs once the slab is finalized.

The usesLeft variable represents the count of uses left for all the outputs. This counter gets decremented after every subsequent input block gets executed. Once the variable has reached 0 the slab is safe to release from memory, and is subsequently freed for reuse for another output block. In practice, the usesLeft variable is calculated at run time, and once it reaches zero, the entire ActivationAtom which owns those outputs is deleted. This implements a cohesive garbage collection system for fine-grain parallelism.

Figure 5.3 - A different perspective on the slab in the slab allocator. Input 1 and Input 2 are pointers to outputs of another activation record(block)

Figure 5.3 displays another view of the memory including the location of input pointers. These input locations contain pointers to output locations as described in Figure 5.2. A collision cannot occur involving the pointers for input 1 and input 2. This collision is avoided because the memory addresses for input 1 and input 2 are known at creation of the block, even if the values are not computed. These input locations are known at creation of the ActivationAtom, and the addresses to which they point to are accessed at the execution stage.

Getting memory management correct for our runtime was important to have a cohesive system. If we resorted to use locking to solve our problems, it is likely a huge performance bottleneck for memory operations would have ensued. Designing our code to be thread-safe, while avoiding the use of locking, was crucial for creating a successful fine-grained runtime.

5.1.3 Summary of Solution

Wrapping up the ideas of our threading and memory management into a cohesive system, we examine the system as a state machine. Figure 5.4 displays a graphical overview of the different stages an ActivationAtom goes through as it executes. Three activation atoms are shown; the first is a supplier of information to the other two, similar to the examples presented previously. It's important to recognize the different stages of ActivationAtoms before examining a code level example.

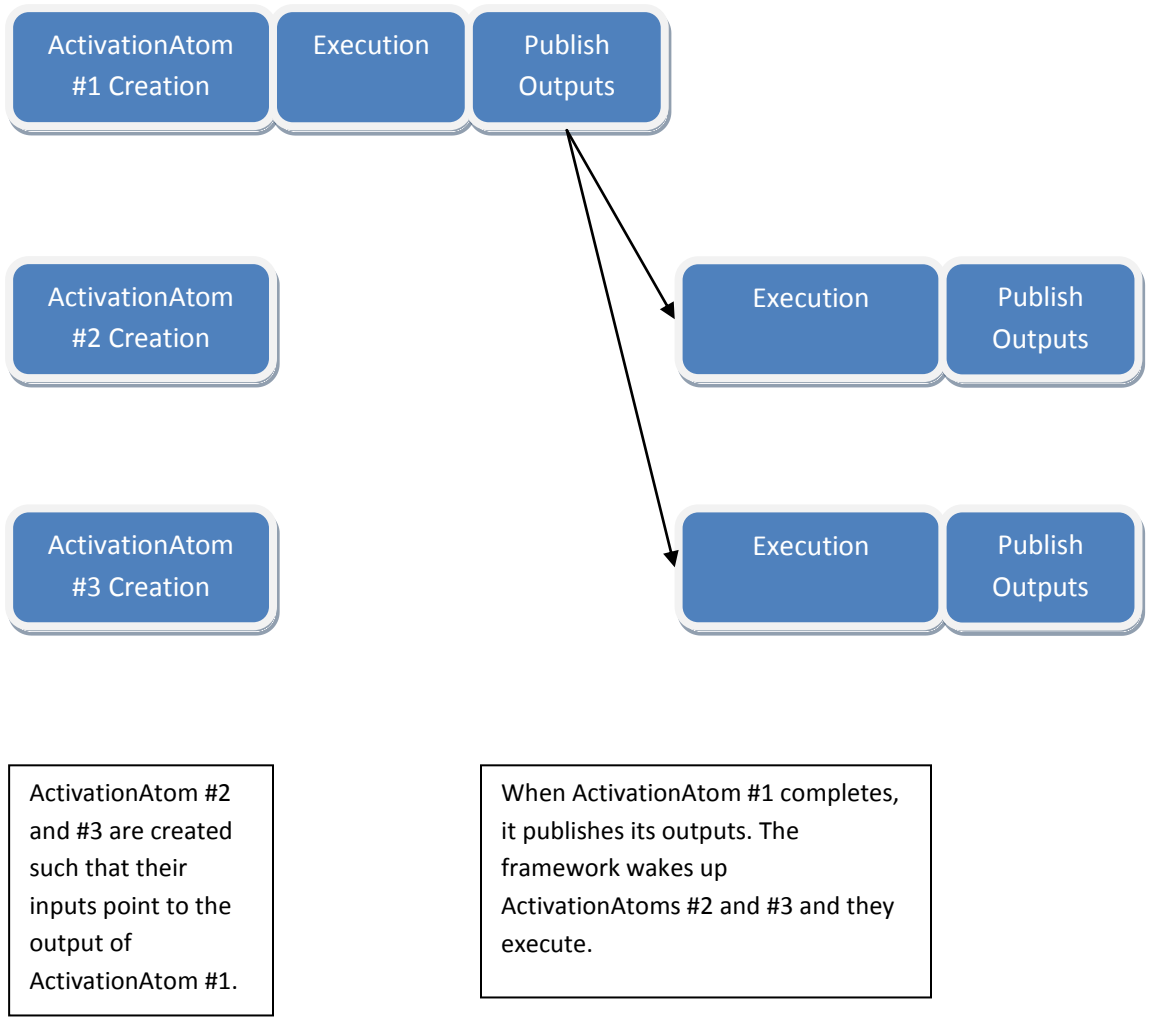

Figure 5.4 – A graphical representation of the execution of our system.

5.1.4 Code Example

It was always the intent to generate code for our runtime, thus manually programming solutions in our runtime is difficult, but not impossible. Outlining and discussing a simple test implementation will fill in the blanks on the work and implementation completed on the runtime.

```
class Supplier : public ActivationAtom
{
public:
  Supplier();
  void process();
};

Supplier::Supplier() : ActivationAtom(0, 2)
{}

void Supplier::process()
{
  int *output1 = (int*)getOutputSpace(sizeof(int));
  int *output2 = (int*)getOutputSpace(sizeof(int));

  *output1 = 2;
  *output2 = 3;
}
```

Program 5.1 – Sample ActivationAtom definition and implementation.

In Program 5.1 we see the definition and implementation of a simple supplier block. In this case the supplier merely publishes a constant to each of its outputs. One should notice that from the perspective of the derived ActivationAtom, Supplier, the needs of publishing an output are abstracted away completely. By using *getOutputSpace()* instead of a traditional *malloc()*, the same functionality to the client is achieved while the framework handles the complexities of fine-grain memory management. In this case, following fine-grain patterns, the ActivationAtom represents a very small operation. Despite this, our runtime is versatile enough to support ActivationAtoms ranging from small operations to large functions.

Besides the definitions of ActivationAtoms, the system needs to stitch the inputs and outputs together of a program's execution so that a proper execution map can be created. This involves syntactically simple C++ code to stitch ActivationAtoms together. The last stage would be to queue the initial 0-dependency blocks into the scheduler, so that those execution blocks without inputs are executed first, as they represent the beginning of the program's execution. Our runtime uses the template design pattern to hook into any application that defines a function with the signature *void run().*

```
void run()
{
  Supplier * ablock = new Supplier();
  Adder * bblock= new Adder();
  Printer *pblock = new Printer();

  //the overloaded new operator requires a parameter.
  //this parameter represents the block that "owns"
  //this InputOutput object.
  InputOutput *a = new(ablock) InputOutput();
  InputOutput *b = new(ablock) InputOutput();
  InputOutput *c = new(bblock) InputOutput();

  //Bidirectional link between blocks and outputs are one function.
  ablock->addOutput(a);
  ablock->addOutput(b);

  //Bidirectional link between blocks and inputs are one function.
  bblock->addInput(a);
  bblock->addInput(b);

  bblock->addOutput(c);
  pblock->addInput(c);

  System::get()->Scheduler->QueueNewFiber(ablock);
}
```

Program 5.2 – C++ code that represents "stitching" of ActivationAtoms

While we always intended to generate code for our runtime through a generator, the generated code is easy enough to follow by hand. This allows us to debug generated code and the runtime easier. Furthermore, it allows a person to manually create an ActivationAtom-based application.

5.2 Summary

The runtime library for our system is a versatile implementation of fine-grain parallelism. It was designed to allow us to test the feasibility of using fine-grain parallelism in a prototype fashion. Ultimately, it was a successful endeavor that allowed us to explore the nature of implementing a fine-grain parallelism runtime and serve as a basis for evaluating algorithms against a specific fine-grain parallelism implementation.

6 Code Generation

The code generator has the important role of linking the graphical front-end and the fine grain-runtime together. The code generator takes the IF1 code generated from SISAL code and turns it into C++ code which uses the fine-grain runtime. This section covers the entire process of generating C++ code from IF1 and tries to give the reader a good understanding of it. In depth detail on the IF1 language and how it is represented by the Code Generator is provided as well as the details on how code is actually generated. Also included in this section are examples on how the fine-grain runtime API is used to increase parallel performance of the original IF1. The code generation process is broken up into three different phases: parsing, linking, and code generation, all of which are thoroughly explained.

6.1 Parsing Phase

Parsing is the first phase of the code generation. An IF1 file is read into memory so that the information can be interpreted and used for generating C++ code for our fine-grain parallelism runtime. The parser framework implements the key components of the IF1 language so that it can be further optimized to achieve a greater degree of parallelism. The main components of IF1 that will be represented by the framework will be simple nodes, compound nodes, edges, literal edges, types and graphs. After the parser creates the objects they will be linked together to create a number of virtual (refers to any item stored in physical memory) graphs which represent the IF1 program. These graphs will then be translated into C++ code that makes use of the fine-grained runtime.

6.1.1 IF1 Background

The IF1 language is based on the concept of directed acyclic graphs (DAG). The simple definition of a DAG is that it is a collection of nodes connected by series of directed edges. IF1 implements this concept though six components: simple nodes, compound nodes, edges, literal edges, types, and graphs. Nodes can be either simple or compound; simple nodes denote simple operations such as add or divide, compound nodes provide complex operations like if statements and looping. Edges connect node outputs to node inputs. Types are used to define the type of data (i.e. int, double, etc…) passed or handled by nodes, edges, and graphs. Graphs are composed of nodes and edges, and are IF1's version of a function.

For example, a graph for the expression (a+b)/5 would be represented by:

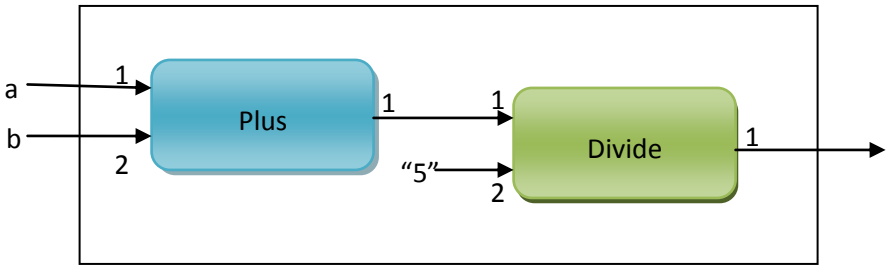

Figure 6.1 - Graph of (a+b)/5 [31]

Figure 6.1 consists of a local graph that has two nodes (represented by the boxes), four edges (represented by the lines), and one literal edge. The box surrounding all the components is the graph

containing the nodes and edges. Nodes can have node inputs resulting from other nodes (“a” and “b”) or they can have predefined constant inputs such as “5” (a Literal Edge).

6.1.2 Types

Table 6.1 - IF1 Types [31]

Entry	Code	Parameter 1	Parameter 2
Array	0	Base Type	
Basic	1	Basic Code	
Field	2	Field Type	Next Field
Function	3	Argument Type	Result Type
Multiple	4	Base Type	
Record	5	First Field	
Stream	6	Base Type	
Tag	7	Tag Type	Next Tag
Tuple	8	Type	Next in Tuple
Union	9	First Tag	

Table 6.2 - Basic Type Codes [31]

Type	Code	Type	Code
Boolean	0	**Integer**	3
Character	1	**Null**	4
Double	2	**Real**	5

The IF1 language supports a broad range of types as shown in Tables 6.1 and 6.2. Types are used by many of the different components (i.e. edges, nodes, graphs) of the IF1 language. Any component that has a type field uses one of the types from Table 6.1. The parser framework represents these types using the *Type class*. Types have global scope and are identified with a unique label in IF1 so they can only be defined once. Any component in IF1 that has a type uses the label assigned to the Type object to reference which type the component uses. IF1 also supports many types (i.e. Tag, Tuple, Union, etc…) that are not present, or have different behaviors then in C++.This means that they have to be converted into a C++ equivalent during code generation, which is discussed later in this section.

6.1.3 Nodes

Table 6.3 - Example Nodes [31]

Node	Kind	Inputs	Outputs	Sub-graphs
Plus	Simple	2	1	0
AElement	Simple	2	1	0
AScatter	Simple	1	2	0
RBuild	Simple	Any	1	0
LoopB	Compound	Any	Any	4
Select	Compound	Any	Any	>2

IF1 nodes represent an operation such as adding, subtraction, or multiplication. There are two different types of nodes, simple nodes and compound nodes. Simple nodes have only a label, which is used to identify the node, and an operation. Compound nodes are more complex and are used to represent complex operations such as loops and if statements. Compound nodes in addition to having a label and operation contain a number of sub-graphs, (discussed in more detail in the graph section). The compound node uses its sub-graphs to carry out the individual parts of the complex operation such as checking the conditional on a loop statement. Therefore a compound node could have any number of sub-graphs each of which have nodes and edges of their own making the operation fairly large and complex. Table 6.3 gives an example of a few different types of nodes and specifies the number of inputs, outputs, and sub-graphs for each node.

In IF1 node inputs and outputs are referred to as ports. Ports are numbered beginning at; the numbering is not shared between inputs and outputs so a port number N can refer to both an input and an output. This shared numbering needs special consideration when connecting nodes and edges in the linking phase. Input ports have a 1 to 1 relationship with their connected output port which means that a port that serves as an input to a node can only have one node supplying that input. Output ports on the other hand have a 1 to N relationship with their connected input ports, which means a single output port can be the source for multiple input ports.

The parser framework implements the nodes concept using the *Node* class. The *Node* class contains a label, operation, list of input ports, and a list of output ports. Two additional classes *SimpleNode* and *CompoundNode* extend off *Node* in order to add the additional functionality needed by the two types of nodes in IF1.

6.1.4 Compound Nodes

IF1 uses compound nodes to represent complex operations. There are five different types of nodes that are defined in the IF1 documentation. Select, TagCase, ForAll, LoopA, and LoopB. Of these we focused primarily on Select and ForAll. TagCase is documented on pages that are missing from our IF1 documentation and has been ignored. In addition to the fact that part of the TagCase documentation was missing, it is only used in handling the Union and Record data structures built into SISAL. The implementation of these data structures was deemed less important than that of other, more fundamental language features. Ultimately, they were not implemented by us, making TagCase irrelevant.

Select –The Select compound node contains at least three sub-graphs. One of these sub-graphs is the selector, and the remaining ones are the alternatives. The selector sub-graph is responsible for processing any input to the compound node and returning a number between 0 and N-2 where N is the number of sub-graphs in the compound node, meaning there are N-1 branching alternatives. This number is then used to select one of the alternatives, which is provided with all of the inputs to the compound node. The result of this sub-graph is passed as output from the compound node. Although the Select compound node is defined as a general purpose switch statement, SISAL only takes advantage of this as an if-then-else statement, using only two branching alternatives.

ForAll – The ForAll compound node provides a method of applying a static set of operations on every value on an input list. The compound node is made of 3 pieces: the Generator, Body and Returns sub-graphs. The Generator sub-graph is defined to run first and is responsible for returning the list or lists of values that will be iterated over. It would then be the responsibility of the runtime the IF1 is being compiled to, to detect the size of these lists and create an appropriate number of Body sub-graphs. Each of the Body sub-graphs processes one value from each of the lists returned by the Generator graph. In turn the output from these functions is placed in an array and passed to the Returns sub-graph. Output in the Returns sub-graph represents output for the rest of the compound node.

LOOPA/B – Although technically different, LoopA and LoopB share much in common. Both types of compound node are made of four components, the Initialization, Test, Body, and Returns. The Initialization and Returns sub-graphs provide much the same functionality that they do in the ForAll Node in that they setup the loop and return its values. The biggest difference in a LoopA/B compound node is its inclusion of the Test sub-graph. This sub-graph is run to check if the loop will perform another iterative execution of its body sub-graph. In LoopA the check is performed after the initial run of the body and after each iteration thereafter. The LoopB Test sub-graph is setup to run before the initial run of the body and before each iteration thereafter. Unlike the ForAll node the LoopA/B nodes are defined to run their Body sub-graph 0 or more times.

6.1.5 Edges

Table 6.4 - Edge Format [31]

Source		Destination			
Node	Port	Node	Port	Type Reference	String/Comment
0	1	4	1	Entry for Integer	Name is “a”
0	2	4	2	Entry for Integer	Name is “b”
4	1	5	1	Entry for Integer	
		5	2	Entry for Integer	Value is “5”
5	1	0	1	Entry for Integer	

IF1 supports two different types of edges, literal edges and normal edges. Literal edges supply a constant value to a node’s input port. A normal edge simply connects the output from one node to the input of another. Both types of edges specify the type of data the edges handle. Table 6.4 shows both literal edges and normal edges. A normal edge has a source node and port as well as a destination node and port. A literal edge has just a destination node and port and uses the string /comment area for the constant value.

The parser framework implements the edges using the *Edge* class which has a variable for storing the edge type and variables for storing both the source and destination nodes. Literal edges (class *LiteralEdge*) and normal edges (class *NormalEdge*) both extend off *Edge*. Literal edges ignore their source nodes and add the additional option of supplying a constant value to the destination node.

6.1.6 Graphs

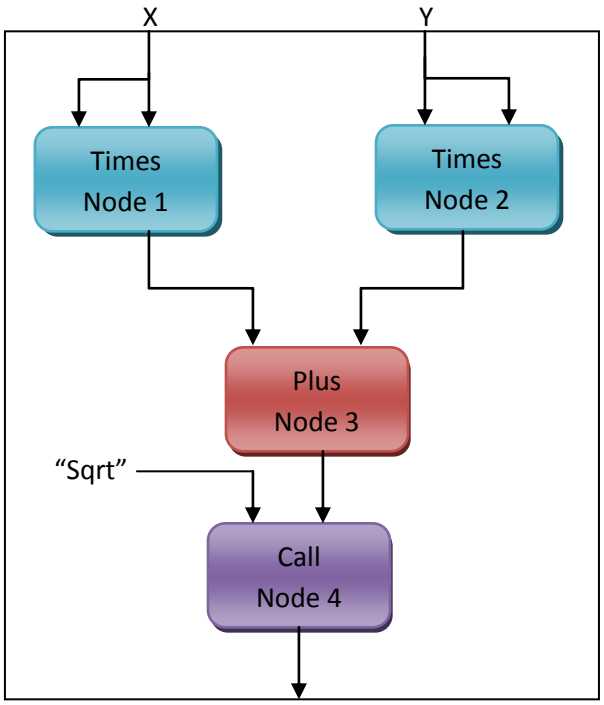

Figure 6.2 - A Local Graph [31]

IF1 uses graphs to represent functions or other logical code groupings. Graphs, like functions in most languages, have global scope and local scope depending on where and how they are declared. For instance in IF1 graphs can be declared three different ways, as local graphs, as global graphs, or as imported graphs. Local graphs are usually declared as sub-graphs within a compound node, but they can also be declared within the global scope of the IF1 program. Imported and global graphs are both declared within the global scope of the IF1 program. Any graph with global scope must have a name assigned to it so that it can be referenced by other sections of the IF1 program (i.e. the function name). Figure 6.2 gives an example of a local graph that could be contained within a compound node. The local graph is built up of four simple nodes that are used in series to produce the final result of the local graph. Currently the Code Generator does not implement *Imported* graphs, which allow you to import functions defined outside the IF1 file.

A graph is only built up of nodes, both simple and compound, and edges; it cannot contain other graphs unless they are a sub-graph within a compound node. Each graph has its own set of nodes and edges that can only be accessed by the graph in which they are declared. For example a sub-graph within a compound node that is declared within a global graph cannot access the global graph's nodes and edges, likewise the global graph cannot access the sub-graph's nodes and edges.

Each graph also has one node that is referred to as node zero. Node zero is the start point and endpoint for all graphs. This node holds all the parameters (inputs) and results (outputs) for a graph and

is the only way another node can interact with a graph. Any results a graph might calculate will always be placed inside node zero for other nodes to retrieve. Node zeros have a theoretically unbound number of inputs and outputs; these must be accounted for when C++ code is generated.

The parser framework uses the *Graph* class for all three types of graphs. Since all graphs are essentially built the same way they can be parsed and stored in the same type of object. The scope of each graph will be the only difference between the three types. Local graphs declared within a compound node will only be seen by that compound node. Global graphs, imported graphs, and local graphs declared outside of compound nodes are treated with global scope and are stored within IF1Root's global space, variable space within IF1Root, so that they can be accessed by the entire IF1 program. IF1Root is an object that is an abstraction of the Code Generator that contains all parts of an IF1 File read into memory.

6.1.7 Parsing an IF1 File

The IF1 file structure makes parsing an IF1 program a simple task once the basic rules about the IF1 language are understood. The most general rules of IF1 are:

- All components (except Type) have a Type.
- Edges, Nodes (compound and simple), and Types all have a label that identifies them.
- Each line in an IF1 file has only one declaration.
- Each line is identified by a single character label (C, X, G, I, {,}, N, E, L, T), which determines what the line represents.
- After a declaration each line may contain Comments or Meta information.

An IF1 file will have all types that are used in the program declared first, followed by graphs that have global scope. The only components of IF1 that have global scope are types, and graphs. The IF1 parser will first parse all types and store them in a map within the global space of IF1Root using the Type's label as the key for the map item. This allows for any IF1 component that has a type to directly map to a Type object when parsed. Once the types are parsed the graphs in the Global scope of the IF1 program are parsed.

The parsing of graphs in IF1 can range from simple to relatively complex. If all a graph contains are simple nodes and edges parsing the graph is nothing more than adding all the nodes and edges to a list, but if the graph contains a compound node it becomes a bit more complex. Compound nodes are built up of multiple sub-graphs, which are just like any other graph in that they have simple nodes, edges and possibly compound nodes more with sub-graphs. Whenever a compound node is reached it will be added to the current graph's node list and then the parser will parse the compound nodes contents before parsing the rest of the graph. After the parsing of all the global graphs is finished, the Code Generator moves onto the linking phase where a virtual copy of the IF1programis made from the parser framework components.

6.2 Linking Phase

At the beginning of the linking phase there is a collection of IF1 components, which consists of simple nodes, compound nodes, edges, literal edges, graphs, and types, in memory with no established

relation to each other. In the global space there are Types and Graphs that have no meaning at this point. At the graph level there are possibly hundreds of nodes and edges that have yet to be linked together. The linking phase of the compilation process is what connects all of these scattered components. Figure 6.3 gives a good overview of general scope of each component at this point.

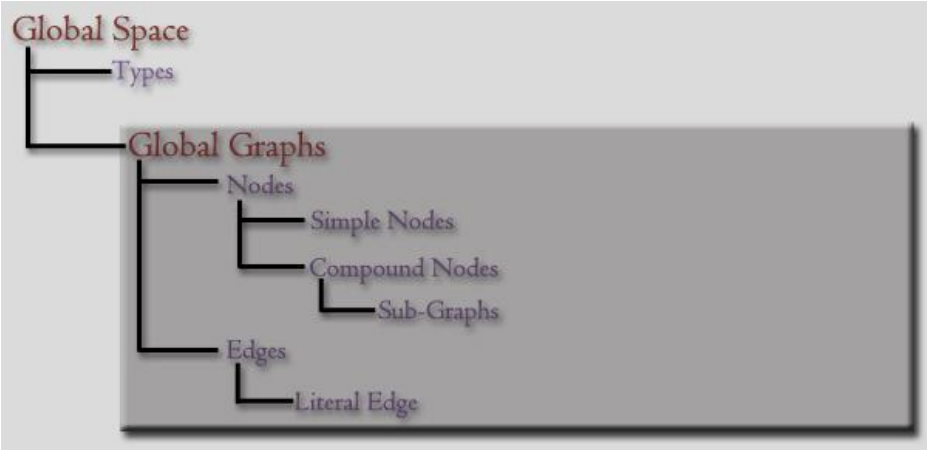

Figure 6.3 - Component Scope Overview

The linking phase takes all the of the IF1 components that were generated in the parsing phase and starts linking them together the end results is a set of virtual graphs which can be used to generate the C++ code; one virtual graph per IF1 global graph. This process mainly involves connecting edges with their respective source and destination nodes. At this phase no links between graphs are made, meaning nodes outside of another graph's scope aren't linked with that graph's node zero.

The linking process begins with the first global graph contained in IF1Root. The process is essentially the building of graphs since they don't virtually exist yet; all that exist inside a graph are list of nodes and edges where nothing is connected. In order to build a graph the list of edges within a graph is sequentially gone through and each edge is connected with its source and destination node. Nodes are identified by their node label which is unique to the current local scope. When connecting an edge to a node the edge specifies whether it supplies an input to the current node or if it uses the current node as an input to another node. This distinction between input and output type is important when generating code for the node since a single port number N can be an input or output.

In the early design phases of the *Node* class, originally it was decided that references to edges in a map using a combination of the edge's port number and input output type as the key. However this design fails because a node's output port has a 1 to N relationship. Edges can use the same output port as a source of input for multiple nodes which means if it was stored in a map any previous value stored by the key would be overridden. This led to a design change of using a linked list to store edges allowing for an output port to be the source of input for any number of nodes.

During the linking phase literal edges are treated just like edges except that they have no source. The same idea applies to simple nodes and compound nodes in that they are treated the same when connecting edges except that compound nodes will also have their individual sub-graphs built

during this phase; sub-graphs are built using the same method as global graphs. When a compound node is found it will put a halt on building the current graph and the compound node's sub-graphs will be built first so that any dependencies on the current compound node are met.

This process is repeated for every graph in the global space until all the graphs are built. At this point only inter-graph linking remains to be done. This inter-graph linking is achieved during the code generation phase where the graphs are finally linked together and the C++ code is generated for the virtual IF1program.

6.3 Code Generation Phase

The code generation phase is the most important phase of the compilation process; it's the phase where most of the work is done. The code generation phase is more than just generating code from the virtual graphs; it has to adapt the IF1 runtime model into the fine-grain runtime model. IF1 has a runtime model that allows for interrupts and blocking where as the fine-grain runtime model does not. This means abstractions such as function calls that relying only blocking have to be turned into a system which does not need blocking which is one of the tasks the code generator carries out. It main job however is generating code that is highly parallel using the fine-grain runtime. This section gives an overview of the code generator which carries out the code generation process.

6.3.1 The Runtime API

The code generator makes use of the fine-grain runtime API to generate code that is highly parallel. The code generator mainly uses the ActivationAtom and InputOutput classes provided by the runtime API for generating parallelizable code. The ActivationAtom and InputOutput classes, as previously discussed in the Runtime Implementation section, provides a way for executing code in a thread safe manner without having to worry about thread safety at the user level. The code generator makes use of this by creating classes that extend off ActivationAtom that implement almost every operation nodes can perform in IF1 and uses InputOutputs for all variables declared in the generated C++ code.

6.3.2 *CodeGen* - The Core of Code Generation

The core of the code generation lies within the class *CodeGen*. *CodeGen* provides all the functionality for generating the fine-grain runtime C++ code based of off the virtual IF1 program contained within IF1Root. In truth, *CodeGen* is a static class that serves as an interface to the many pieces that that make up the code generator. The functionality that *CodeGen* provides is the following:

- Random name generators for objects, variables, and literal values.
- A function that generates code for all operations.
- Functions for generation InputOutputs and ActivationAtoms.
- A function that generates code for all IF1 global functions.

One of the key pieces of functionally that *CodeGen* provides is the ability to generate code for all node operations. This was accomplished by establishing a standard for code is generated for a node. As

previously mentioned all node operations have an ActivationAtom created for carrying out the operation. All operations in IF1 have a designated constant number to represent them which every node has. This allows us to associate every ActivationAtom with a number as well which lead to creation of generic function (Program 6.1 below) that all nodes could use for code generation.

```
generateOpCode(GeneratorStream* streams,
               NodeOperation op,
               VarList* inputs,
               VarList* outputs,
               Node* node)
```

Program 6.1 – Generic Function for a Node Operation Code Generation

GenerateOpCode() provides everything need for generating code for any node operation. The first parameter provides the streams for output (these streams discussed in more detail in the next section). The next parameter provides the number of the operation the node performs which is used to determine the ActivationAtom to generate. The next two parameters provide the variables for carrying out the operation. For example if the operation was add. The input variables would have the two numbers being added together. The output variables would be the location to store the results of the addition. The last parameter provides the node the operation belongs to.

Another piece of key functionally that *CodeGen* provides is the generation of code for the global graphs in IF1Root. The global graphs in IF1Root will make up the main function in the generated C++ program. This generation is performed when IF1Root calls the *CodeGen* function *generateGlobalFunctions()*, which generates functions for all global graphs in three different three steps. The first generates prototype functions for all the global graphs, the second step generates the runtime *run()* function, and the last step generations the content for the actual functions.

Generating function prototypes is a fairly complex process that relies heavily on the IF1Type information. The IF1 language does not implement functions like a C++ program would (i.e. “<return type>functionName(params ...)”). Since IF1 uses the abstraction graph to represent a function it has to use the graphs Type information for declaring what the parameters and results for the graph are. IF1 basically classifies a function signature as a Type which means it has to be analyzed and converted into C++ format. Types implement functions with the abstraction of tuples which is essentially a list of Types. Therefore the entire lists have to be parsed and concatenated together to create function signatures. Since functions in the runtime can’t block we must add the result parameters to the function signature as well (more detail on how this works in the Call Node section).

In addition to generating the function headers in step one the edges linked with node zero for each graph have to be assigned a variable name. This is because of the fact that when function prototypes are generated the parameters for the function are assigned variable names. The parameters and results of a function are actually the edges connected to a graph’s NodeZero. Edges once translated into C++ represent variable names. Since edges represent variable names and the edges allow access to the function parameters, these edges must be assigned the name that was assigned to the function

parameters. Also since ports for the result of the function have 1 to N relationship, all ports with an edge connected to the same port must have the same variable name.

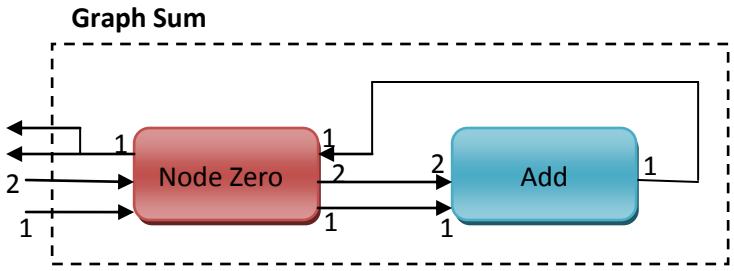

Figure 6.4 – Graph and Edge Example

```
voidFoo(InputOutput* input1, InputOutput* input2,NoOp<int>* output1){
Add<int> adder = new Add<int>();
InputOutput* result  = new (adder) InputOutput();

adder->addInput(input1);
adder->addInput(input2);
adder->addOutput(result);
output->addInput(result);
}
```

Program 6.2 – Graph and Edge Output

Figures 6.4 and Program 6.2 above represent a graph and the code generated for the graph. NodeZero takes in two inputs and has one output, notice the one output is used by two different edges. The code generator will realize this and only generate one variable for the output which can be used as input for any number of nodes.

After the prototypes have been generated*, generateGlobalFunctions()* enters step two where the *run()* function is generated. The *run()* function is the starting function for the fine-grain runtime and it will call the *Main()* function belonging to the IF1 program. The results for the *Main()* function will also be handled and printed out by the *run()* function. The input params (from the command line) for the *Main()* function are first parsed and for each input param a NoOp is created and the value is added as an Input to the NoOp. NoOps are then created for each output that will be generated by the *Main()* function and each NoOp will then have an InputOutput added to its outputs. There are no inputs added to the NoOps used for the results of *Main()* since the inputs will be the results of the *Main()* function. Once all the NoOps to handle the inputs and outputs of the *Main()* function are generated the actual function call to *Main()* is generated. The parameters for the call are the input NoOps output InputOutput variables and the output NoOps (this is discussed in more detail in the Call Node section). After the *Main()* call is generated, PrintValue ActivationAtoms are generated to print out the results of the *Main()* function. An example *run()* function is shown in Program 6.3.

```
1.  NoOp<int,1>inputObj = new NoOp<int,1>();//create  NoOp to hold input param
2.  //create InputOutput to hold value of input param
3.  InputOutput paramIn0 = new(inputObj) InputOutput();
4.  inputObj->addInput(paramIn0);
5.  ….//Set paramIn0 value here
6.  InputOutput paramOut0 = new(inputObj) InputOutput();
7.  inputObj->addOutput(paramOut0);

//create NoOp to store results of the main function
8.  NoOp<int,1>* resultObj = new NoOp<int,1>();
9.  InputOutput * mainResult = new(resultObj) InputOutput();
10. resultObj->addOutput(mainResult);


11. Main(paramOut0,resultObj); //Call Main

//print results
12. printValue<int>* printV = new PrintValue<int>();
13. printV->addInput(mainResult);
14. }
```

Program 6.3 – Example Run() function

After the *run()* function is generated step three starts which is where all the global graphs have a function generated for them. This step is where most of the time is spent and most work is done. For each global graph a function header is generated using the previously generated function headers in step three to save time. NodeZero for each graph is where the code generation process starts and ends. NodeZero is searched through and generates code for every node in the graph. The process for generating code for a node is covered in the next few sections. Almost every node has its code generated the same way, but there are a few exceptions. Once the functions for all global graphs are generated the code generation process is finished and the code can be compiled and ran.

6.3.3 Generator Stream

A core piece of the code generating process is the *GeneratorStream* class. The *GeneratorStream* is a collection of streams that can output to any location that is assigned to them. All code generator output is done through the *Generator Stream*. Currently the code generator uses to main streams; main code stream and header stream. The *GeneratorStream* class allows for any number of streams to be added that might be needed in future development and is not limited to two main streams. The main code stream is where most of the generated code is outputted to; output is usually to a file that can be compiled and ran. The header stream is mostly used by compound nodes, which require custom classes to be generated (this is discussed in more detail in the Compound Node Code Generation section). The stream output location for either of the main streams can be switched at any time during the code generation process. This provides a lot of flexibility in how the code is generated.

6.3.4 Prepping a Node

Before code can actually be generated for a node a lot preparation has to be made. The preparation is the same for every node (compound and simple). The first thing that has to happen is that all edges connected to the node have to be converted into variables. As previously mentioned edges represent variables once converted to C++. Therefore every edge connected to a port on the node has a random variable name generated for it. In addition every edge is encapsulated in a variable object that provides additional information needed by the code generator such as where the edge is a literal or not, and the type of the edge. One variable name is generated per port, so edges that use the same port will all have the same variable name.

Since NodeZero is always the starting place of code generation when in a graph. Once variables are generated for NodeZero a recursive algorithm works its way through the graph of nodes starting with the node that outputs to NodeZero. This means that the algorithm works backwards through the graph always generating a child's variables before its parents' variables. Once NodeZero or a node with no parents is reached and the variables are generated for it, the algorithm generates the actual code for each node as it works its way back through the graph until it reaches NodeZero. An overview of the actual code generated for each node is covered in the next few sections.

6.3.5 Simple Node Code Generation

As previously mentioned a simple node in IF1 performs only one operation. This means that the code generation for a simple node is relatively the same for all simple nodes. Program 6.4 below is a basic template for generating a simple node.

```
1.  void generateOperation(GeneratorStream* streams,VarList *inputs,VarList *outputs,Node *node)
2.  {
3.  ostream&stream = streams->getMainCodeStream(); //get main stream for writing

4.  string objName;
5.  string classType = (…);//Set classType equal to the ActivationAtom to generate (i.e.) Add<int>

6.  objName = CodeGen::generateObjName();//aquire a random objName
7.  node->name = objName;//assign the node its object name

8.  //write the actual C++ code for the ActivationAtom
9.  CodeGen::generateActivationAtom(stream,objName,classType.append(type.str()));

10. //generate all add input and output code
11. writeAddInputOutputs(stream,objName,inputs,outputs);
12. }
```

Program 6.4 – Template for simple node code generation

The key line in Program 6.4 is line 5 where the class Type is defined. This line determines what type of ActivationAtom is created. Other than line 5 all simple node operations follow the same pattern where

after the class type is defined an object name is generated to hold the ActivationAtom and all inputs and outputs are added to the object generated.

6.3.6 Call Nodes

IF1 supports the functionality of function calls though the implementation of the call node (a simple node). However IF1 built its call node with the intention that when the function is called it will block until a value is return. Blocking is not supported by the fine-grain runtime model and therefore the call node has to be converted into a format that requires no blocking. The solution to this problem is similar to how the Main() function is called from the run() function. Program 6.5 below shows an example of how a call node for the function "int foo(int)" would be generated for the fine-grain runtime model.

```
1.  NoOp<int,1>* resultObj = new NoOp<int,1>();
2.  InputOutput * fooResult = new(resultObj) InputOutput();
3.  resultObj->addOutput(fooResult);

4.  //randomInput is a previously defined InputOutput
5.  Foo(randomInput,resultObj);

6.  printValue<int>* printV = new PrintValue<int>();
7.  printV->addInput(fooResult);
8.  }

9.  void Foo(InputOutput* param0, NoOp<int>*  param1) {
10. Add<int>*  adder = new Add<int>();
11. adder->addInput(param0);
12. ……..
13. //result (an InputOutput) is a calculated value by the function
14. param1->addInput(result);
15. }
```

Program 6.5 – Example of call node implementation

When the C++ code is generated for the call node, instead of writing an ActivationAtom that provides the call node functionality, the function is written like a standard function call in C++ (line 5). The function is always void and includes function parameters for both inputs and outputs (results). The inputs for the function are always first and the outputs always follow. The inputs consist of InputOutput variables that ActivationAtoms within the called function can add as inputs (line 11). The outputs of the function are NoOp's that require the results generated by the function to be added to the NoOp as inputs (line 14) so that ActivationAtoms after the function call have access to the results.

6.3.7 Compound Node Code Generation

Code generation for compound nodes proved to be one of the most difficult challenges encountered. The code generation for normal, non-compound nodes works based on the fact that there exists in IF1 essentially an interconnected web of nodes. Within compound nodes, however, there is merely a collection of isolated sub-graphs with no interconnections. It is up to the implementer handling the IF1 or its equivalent representation to form the appropriate connections among these sub-graphs and the encapsulating compound node itself according to the IF1 specification.

6.3.7.1 Select Node

The Select Node is the IF1 construct that provides general capabilities for branching control flow. While the IF1 specification allows for the equivalent of a "switch" statement in other languages like C, SISAL only implements two-way branching in the form of if-then-else statements. This constraint of exactly two branching options simplifies the implementation of the Select Node for use with SISAL. Also, the SISAL parser apparently always orders the Select Node sub-graphs in the same way, which is not part of the IF1 specification. This further simplifies the implementation of the Select Node for our purposes.

The Select Node contains at least three sub-graphs, and in SISAL it always has exactly three. The first sub-graph is the condition upon which the branching depends. This sub-graph must always produce a single integer as output, the value of which corresponds to a particular branching option. The rest of the sub-graphs are the actual branching options, only one of which will be selected and executed based on the output of the condition sub-graph.

In the IF1 specification, the order of the sub-graphs does not have to correspond at all to the integer values associated with the sub-graphs. Each Select Node in IF1 contains a sequence of numbers indicating which sub-graph is the condition and which integer values correspond to each branch option sub-graph. However, the SISAL IF1 generator always places the condition, or 'if', sub-graph first, the 'else' sub-graph second, and the 'then' sub-graph third. This is because the 'else' sub-graph corresponds to Boolean False, or integer value 0, and the 'then' sub-graph corresponds to Boolean True, or integer value 1. Every sub-graph, including the condition sub-graph, has access to the inputs of the entire Select Node. Every branch option sub-graph must produce the same number and types of outputs, which correspond exactly to the outputs of the entire Select Node. Figure 6.5 illustrates the IF1 representation of a SISAL if-then-else statement. Here the Select Node has sets of N inputs and K outputs, and the three subgraphs are contained within, but they are isolated and not connected to anything.

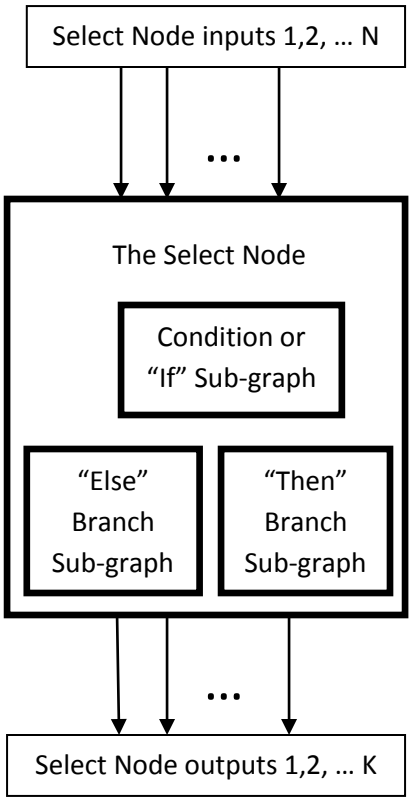

Figure 6.5 - IF1 representation of a SISAL if-then-else statement

To accomplish code generation for the Select Node, a special Selector activation atom class was implemented. A unique class, a child of Selector, will be generated for each Select Node in a given program. The Selector class takes a single integer as input and, based on that input, executes the appropriate runtime code to carry out the selected branch of the Select Node. To do this, the code for each branch option sub-graph is written as the implementation of a method in the Selector class. The code within the condition sub-graph of the Select Node is generated normally in the primary code generation stream. The output of the condition sub-graph is given as input to the Selector activation atom for the Select Node. Based on this input, the Selector then calls the method associated with the selected branch.

Some extra work is necessary to make the inputs of the Select Node as a whole available to each of its sub-graphs. In the case of the condition sub-graph, this can be accomplish simply by taking the compound node inputs and passing them through a set of NoOp activation atoms, which send a value along unchanged, to the nodes of the sub-graph. However, this is not sufficient for providing inputs to the branch option sub-graphs. This is because the Select Node inputs would be made available to every sub-graph at the same time, while the condition sub-graph would need time to execute before the appropriate branch sub-graph could be created and readied for execution. This would create a race condition that could disrupt the flow of data and control in the runtime. As a result, the Select Node inputs are instead passed through a set of DelayedNoOp activation atoms to get to the branch sub-graph. These activation atoms are like NoOps, but they do not make their outputs available until

triggered by the Selector activation atom, after it has readied the appropriate branch sub-graph. This ensures the necessary synchronization so that the data and control flow in the runtime is as it should be.

Figure 6.6 illustrates how the sub-graphs of the Select Node are connected by the code generation process. The inputs of the whole node are provided to a set of input objects, represented by the circles, from which they can be distributed and made available to all of the sub-graphs. Likewise, a set of output objects, again circles, provide the outputs of the whole node to other nodes, and the actual output values will be provided to this set of objects by the branch sub-graph that is selected and executed. The Selector object contains the two branch sub-graphs and executes the one selected by the input received from the condition sub-graph. The only part mentioned above but not shown here is the use of DelayedNoOps. All of the arrows from the input objects to the sub-graphs represent NoOps, either delayed or not. Those which connect to the branch sub-graphs are DelayedNoOps which take additional inputs as synchronization triggers from the Selector object, but including connections for the triggers in this in the diagram would only make it more cluttered and potentially confusing.

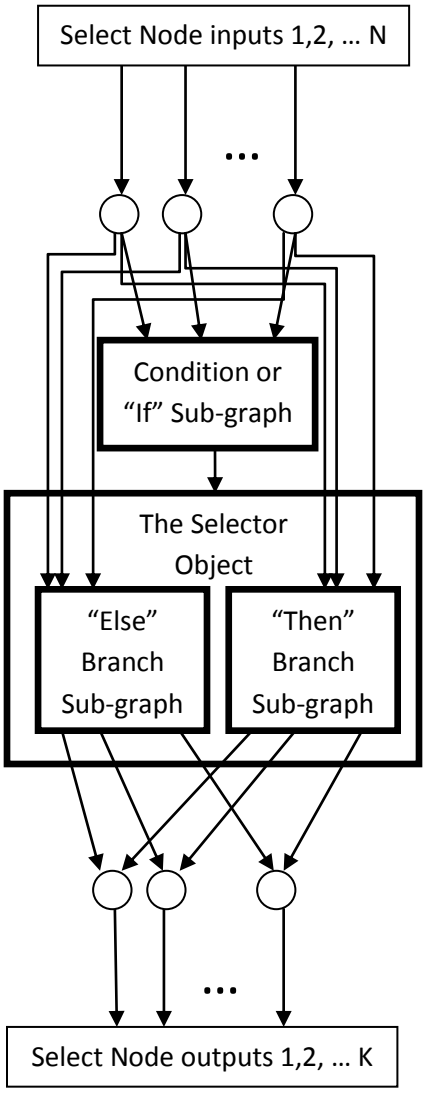

Figure 6.6 - Select Node subgraphs connected by code generation

6.3.7.2 ForAll Node

The ForAll Node provides the facility to run a set of functions over every element in a multiple value array. A ForAll guarantees that no computation will rely on the value of a computation from a previous iteration. This guarantee allows us to run all of the iterations simultaneously. To implement in our runtime we have created three special activation atoms. The MakeMultiple, BreakMultiple, and ForAllRun. MakeMultiple and BreakMultiple perform as their name implies. MakeMultiple takes a set of inputs and creates a multiple value array from them. The BreakMultiple in turn takes a multiple value array and returns one element of it. These are used to compose outputs and decompose inputs respectively. The ForAllRun activation atom is the workhorse of the ForAll node. It is responsible for determining the number of elements in an input array and generating the activation atoms for each value.

As was discussed in the IF1 section, ForAll nodes are composed of three parts: An Initialization sub-graph that is responsible for returning a multiple value array, a Body sub-graph that performs transformation functions on each element of the array, and finally a Returns sub-graph that uses the transformed array to determine the final output. When generating code the same strategies used to generate the sub-graphs for the Select node can be applied to the Initialization sub-graph, and with the introduction of the MakeMultiple the Returns sub-graphs. The Body sub-graph represents a more challenging task.

Our strategy in creating the Body sub-graph of a ForAll node involves the use of node-specific constructor functions. These take inputs, which specify the activation atoms to connect to as input and output. Additionally, an input specifies which element of the multiple values to use as input. The core of the constructor function is also generated in the same manner as the sub-graphs of the Select Node, however this code needs to be wrapped by BreakMultiples, providing the input and runtime calls which connect the inputs and outputs of the body.

The ForAllRun activation atom, as has already been stated, is the most important part of the generated ForAll Node. The ForAllRun is responsible for providing two services. Firstly it calls the Body constructor function once for each element in the multiple value that is generated by the initialization sub-graph. Secondly it provides a way to forward the inputs of the ForAll Node to the Body sub-graphs. The task of calling a constructor for each value is relatively simple. The ForAllRun activation atom must simply look at the size of one of the multiple values (SISAL guarantees that all multiple values used as inputs are the same size) a traditional C++ for loop is then used to call the constructor function once for each value. The task of forwarding inputs is slightly trickier. This is necessary because the body is generated dynamically at runtime. If the activation atoms which provide its input have run to completion before it is generated and connected then their outputs will be released from memory and unavailable to the body. IF1 states that the body sub-graph receives as input all of the inputs to the ForAll node in addition to the results of the initialization sub-graph. Because our Body sub-graphs and Initiation sub-graph share these dependencies it is guaranteed that our runtime will have released from memory the inputs that our body sub-graph needs before it is created. Because our ForAllRun does not need to be dynamically generated it can successfully connect to the inputs to the ForAll Node, it then provides these values unchanged as outputs which the Body sub-graphs can connect to.

Creating a ForAllRun activation atom, which is general enough, to be used for all instances of the ForAll node required a creative mix of templates and Varargs (a C/C++ method for specifying a function which takes an unknown number of arguments of the same type).

6.4 Summary

The Code Generator section covers one of the major time investments of the project. The Code Generator is the intermediate piece that links the graphical front-end and the fine-grain runtime together. SISAL code is generated by the front-end, which is translated into IF1. From there IF1 is passed through the code generator where it is translated into C++ that uses the fine-grain runtime. During the code generator process we change the runtime model from a model that supports interrupts and uses a stack, into one that does not support interrupts and uses the concept of activation atoms. The code generator currently only does a simple translation of IF1 to C++, but further work could be done in optimizing the IF1 code so that it performs better in the fine-grain runtime.

7 Analysis & System Evaluation

In previous chapters, we have introduced the concept of parallel computing and built up the fundamental ideas of our design goals and how those goals translated into the actual implementation of our system. This chapter focuses on the evaluation of the usefulness of the ideas we investigated, specifically fine-grain parallelism, graphical programming, and the runtime model we implemented. We examined the Fast Fourier Transform, H.264 video codec, and Quicksort. Each one of these criteria is evaluated in the scope of a popular potentially parallelizable algorithm. Before being discussed in scope of our system, a background discussing the importance of the algorithm and why we chose it is presented for each algorithm.

7.1 Fast Fourier Transform

To Richard Lyons "Discrete Fourier transform (DFT) and the corresponding Fast Fourier Transform (FFT) is one of the most common, powerful procedures in the field of Digital Signal Processing." [38] The DFT is a discrete version of the continuous Fourier transform. A continuous Fourier transform is used to convert a signal from the time domain to the frequency domain. A DFT or FFT is used to convert a sampled signal from the time domain into the frequency domain. The Fast Fourier Transform will be shown to fit well into our notion of fine-grain parallelism. Because of this, we explore the feasibility of representing the FFT in our graphical and runtime models.

7.1.1 Discrete Fourier Transform

The continuous Fourier transform equation is converted to a summation to form a Discrete Fourier transform. This technique is similar to estimating a continuous integral with a Reimann Sum. Both the discrete and continuous transforms convert a signal from the time domain to the frequency domain.

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j\,2\pi f t dt}$$

Equation 7.1: Continuous Fourier Transform

$$X(M) = \sum_{n=0}^{N-1} x(n)e^{-j\,2\pi n m\ /N}$$

Equation 7.2: Discrete Fourier Transform

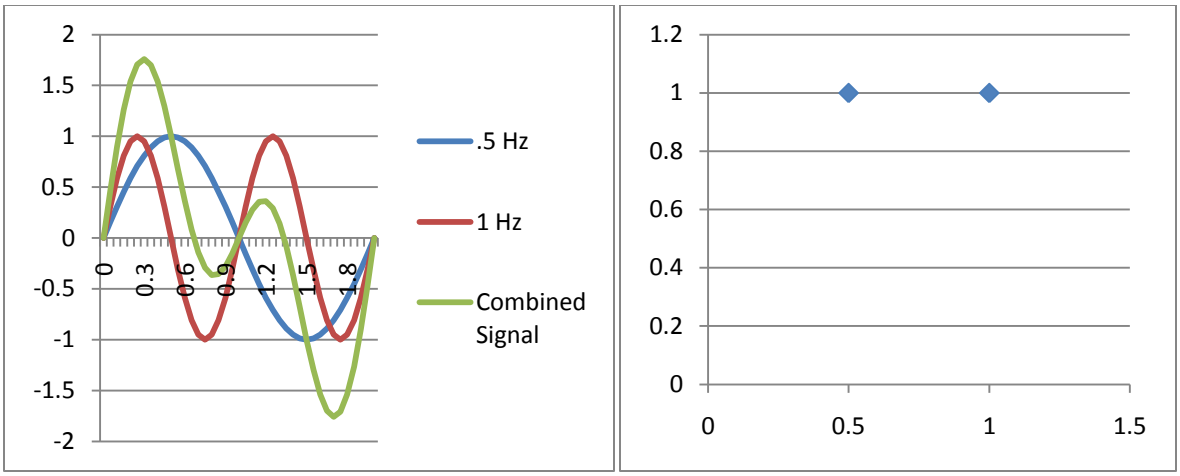

Figure 7.1 – (left) Displays the time-domain of a ½ hz signal and the combined 1 hz signal. (right) Graphically represents the result of the FFT of the combined signal.

The DFT is straight forward to implement on a digital computer. Because of Euler's relation $e^{-j\theta} = \cos(\theta) - jsin(\theta)$ [38] we can calculate what is known as the twiddle factors. These twiddle factors are the "munchkins" that are each multiplied by the input samples in the buffer to compute the DFT. Depending on the value of N chosen, we have a larger or smaller frequency resolution. To figure out the frequency resolution of an FFT or DFT one looks at the band spacing by dividing the sampling rate by N. This gives you the fundamental frequency of the FFT, and each index position of X(m) is a multiple of that frequency.

The DFT has an algorithmic complexity of O(n^2). This is because each summation from Equation 7.2 is done N times. O(n^2) is also referred as having polynomial complexity. Traditional rules of limits can be used to compare two different algorithms if both of their complexities are known, to compare their performance at edge conditions [38].

7.1.2 Fast Fourier Transform

Due to the DFT's high algorithmic complexity it can be unwieldy for small microprocessors to deal with large input sizes. In 1965, Cooley and Tukey described a divide and conquer algorithm to efficiently calculate a discrete Fourier Transform called the Fast Fourier Transform [38]. Interestingly the original author of the FFT was German Scientist Johann Frederich Gauss. While Guass implemented the FFT a hundred years before the implementation of the digital computer, the computational value of it was overlooked until Cooley and Tukey reinvented it in 1965. This doesn't lower the achievements of these two engineers as much as express the importance of the Fourier Transform in modern science.

The FFT is a divide and conquer approach to solving the DFT. It is not an approximation, it solves the same program as a DFT just in less cycles. Other applications of the divide and conquer algorithm include quick sort, parsing, and multiplying large numbers. In general D&C algorithms can be expressed with some level of recursion. Depending on the compiler, the recursion may be removed to express the problem in a faster way to better exploit the inherent parallelization of D&C algorithms [38].

The FFT works mathematically by splitting the DFT in half into two new DFTs. It then does the same until all that is left is a series of 2-point DFTs. At this point, those values are calculated and bubbled back to compute the final N-point FFT [38].

7.1.3 Implementation

```
void btds_fft_r(Complex *x, const Complex *w, short N)
{
        int i; //loop counter variable.
        Complex new, temp;
        /**
        At the lowest level compute a 2-point DFT.
        */
        if (N == 2)
        {
                MPY_COMPLEX(temp, w[0], x[1]);
                ADD_COMPLEX(new, x[0], temp);
                MPY_COMPLEX(temp,w[1*BUFFER_SIZE/N], x[1]);
                ADD_COMPLEX(x[1], x[0], temp);
                x[0] = new;

        }
        else
        {
                btds_fft_r(x, w, N/2);
                btds_fft_r(x+(N/2), w, N/2);

                /**
                Once the left and right sides are done, mash them up as described
                in the butterfly diagram.

                This takes N/2 operations because we push and pull from each
                data line.
                */
                for (i = 0; i < N/2; i++)
                {
                        MPY_COMPLEX(temp, w[i*(BUFFER_SIZE/N)], x[i+N/2]);
                        ADD_COMPLEX(new, x[i], temp);
                        MPY_COMPLEX(temp,w[(i+N/2)*(BUFFER_SIZE/N)], x[i+N/2]);
                        ADD_COMPLEX(x[i+N/2], x[i], temp);
                        x[i] = new;
                }
        }
}
```

Program 7.1 - Implementation snippet of an FFT in C.

Program 7.1 is straightforward recursive C code representing an FFT calculation. The FFT being a divide & conquer algorithm, it has strong potential for parallelization. It is a good test to see how our runtime language effectively maps to common non-trivial computations. The C solution implements the butterfly diagram from right to left, but the activation atom approach would implement it from left to right, following the arrows. This paradigm change is due to the switch between procedural and dataflow languages. While seemingly minor, it is important to mention: implementing the algorithm in block-like notation seems to follow traditional graphics better than procedural C code.

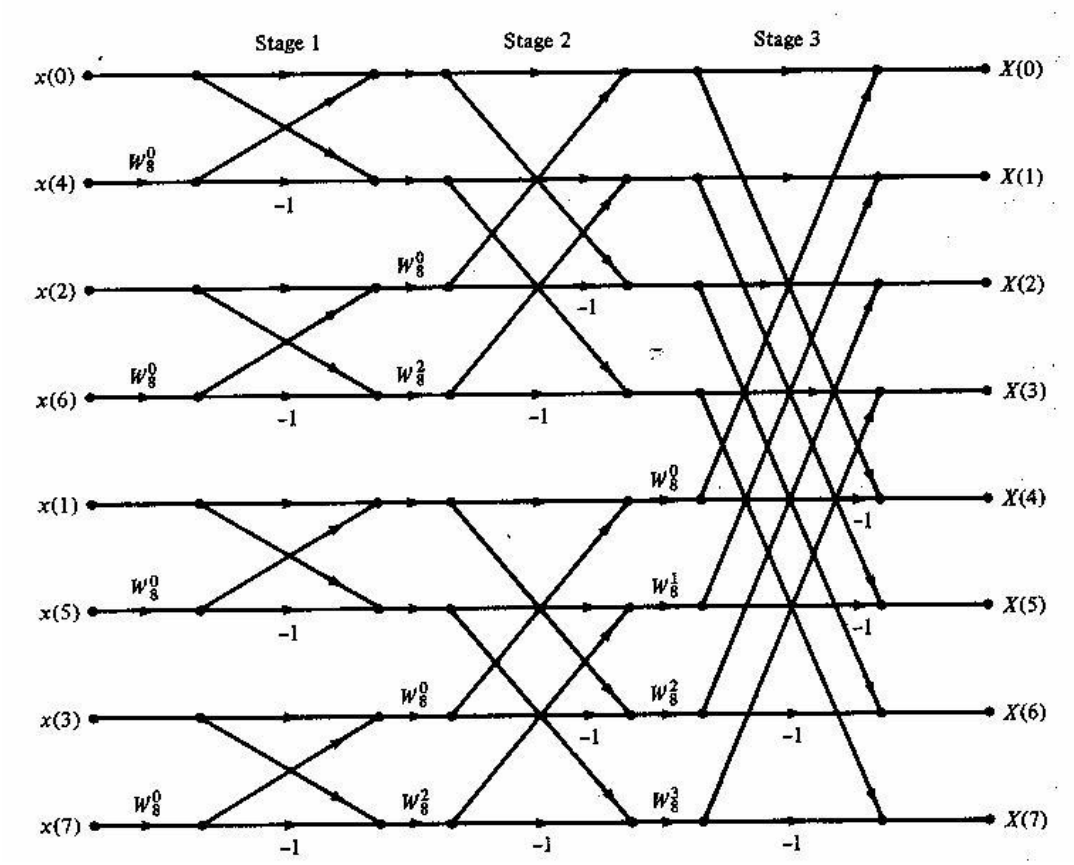

Figure 7.2 - FFT butterfly diagram for an 8-point FFT – x(n) refers to the inputted signal, X(n) refers to the outputted signal. [38]

By looking at Figure 7.2, we can see the natural parallelization of the FFT. The disadvantage to the block model for our block-based runtime is that the entire system needs to be created to execute the FFT. That is each block needs to be constructed and hooked together before execution begins for each FFT computation. This is because there is no elegant way for nodes to create another node if input/output connections need to be made. Nevertheless, implementing the FFT in pure terms of the butterfly diagram would look something like the following:

1) Create execution map of activation atoms. For an FFT of order 8, there should be 4 2-point DFT blocks.
    a. Each 2-point DFT outputs to the inputs of a one of 2 4-point DFTs.
    b. The 4 point DFTs output to the 8 point final DFT calculation.
2) Queue the first 4 2-point DFTs to the scheduler.

While this implementation would allow for concurrency, it is coarse level of parallelization on par with conventional techniques. The first stage having the most concurrent calculations would exhibit the most parallelization. The next stage would exhibit half that, the next another half, until the final N-point DFT which would exhibit no parallelization! The entire notion of fine-grain parallelism is to extract "hidden" parallelization that the human programmer would not normally look for. Because our system

never was fully completed we can only reasonably assume what our system would have found foundations for assumption. Using the logic we planned for it we can extrapolate a more concurrent solution for the FFT. Fortunately the butterfly diagram for the FFT provides a coherent graphical representation of the problem. In lines with our project hypothesis, this graphical representation, while not our system's representation, makes concurrent opportunities more apparent.

7.1.4 Fine-Grain Analysis

Recognizing each intersection is an operation shows the true parallel nature of the FFT. If each ActivationAtom implements this intersection, a constant amount of parallelization is achieved for the problem. In fact for an N-point FFT, N concurrent operations could be occurring at each stage. By shrinking the ActivationAtom abstraction size from the level of an FFT stage or C-function to an operation, we have increased the potential parallelization by many orders of magnitude. While the amount of parallelization for the fine-grained approach is constant at order of the FFT (N), the amount of maximum parallelization for the course-grained approach can be calculated mathematically based off of the order of the FFT.

$$P = \frac{\sum_{i=1}^{\log_2(N)} \frac{N}{2 * 2^{i-1}}}{\log_2(N)} = \frac{\sum_{i=1}^{\log_2(N)} N * 2^{-i}}{\log_2(N)} = \frac{N}{\log_2(N)} * \sum_{i=1}^{\log_2(N)} 2^{-i} \cong \frac{N}{\log_2(N)}$$

Equation 7.3: The average number of concurrent operations per stage of a course-grain parallel N-point FFT

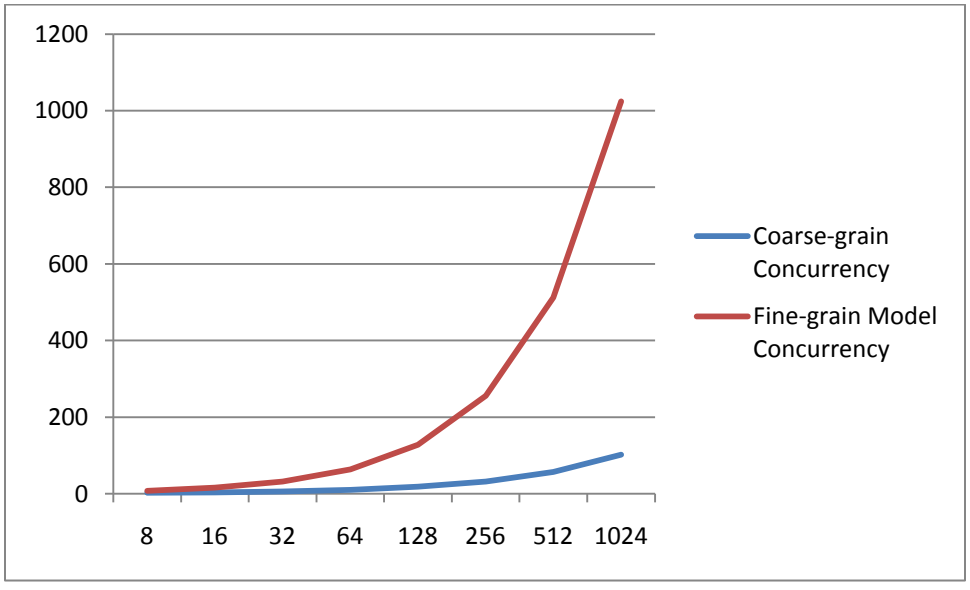

Figure 7.3 - A graphical comparison of asymptotic behavior of fine-grained vs. coarse-grained parallelism for the FFT.

Given the concurrency approximation for the coarse-grain approach, we see the fine-grained approach provides a huge benefit in terms of yielding a maximum in parallelization for the FFT. Numerically, using fine-grain parallelism for a very large order FFT of 1024, a constant number of 1024 concurrent operations can be executed for a total of 10 stages. Under the coarse-grain approach only an

average of 102 concurrent operations can be run for each stage, more at the beginning but less and less as the FFT nears completion. Unfortunately, no modern CPU hardware is capable of running such a large number of concurrent operations in either scenario for large order FFTs. Therefore it should be of interest to the hardware community to increase the number of concurrent ALU or FPU operations allowed on CPU hardware. Due to this massive amount of parallelization the use of a sequential CPU for the task may be improper and the environment would be better using stream-based processors such as a GPU. There has been research to utilize general purpose GPU hardware for computing the FFT. [39] The impact of this on the potential direction for our runtime is elaborated in the Future Work chapter.

7.1.5 Runtime Representation

Because the FFT was analyzed to fit well into the notion of fine-grain parallelism, it is a good example to evaluate our runtime model in terms of implementing a real-world algorithm. Attempting to implement the FFT calculation in our runtime is straight forward. Avoiding the calculation of the twiddle factors, which involve a one-time overhead in terms of a series of trigonometric functions, we concentrate on the runtime code equivalent to the C code presented earlier.

There are two operations that are required for the FFT; 2-ary multiplication and addition. These both can be implemented in our runtime. The rest of the algorithm is implemented by stitching these operations together in the correct order.

```
template<class T>
void Add<T>::process()
{
        T a = *((T*)inputs.next()->argument);
        T b = *((T*)inputs.next()->argument);
        T* c = (T*) getOutputSpace(sizeof(T));
        *c = a + b;
}
```

Program 7.2 - Representation of addition in our runtime. Multiplication is nearly identical until the last step.

The stitching of the FFT operations is the more complicated operation. Because our runtime does not allow adding outputs to blocks dynamically, it makes it harder to implement an N-point FFT like the provided C example. The problem arises when an ActivationAtom intends to create new ActivationAtoms to supply inputs to already existing ActivationAtoms. This came about both due to our non-interruptible block model of implementing functions in two halves and abstractions we made in the runtime library itself. Specifically, it is not visible for one ActivationAtom to see what ActivationAtoms it is supplying. This abstraction was not intentional, but happened after finishing the implementation of fine-grain parallelism. The runtime was tested and designed for a system when all blocks are known ahead of time, not for a more dynamic execution model.

Fortunately, while these shortcomings make implementing the FFT in our runtime more difficult, but not impossible. By going down a layer and utilizing the underlying C++ functionality of our runtime, we can supply the necessary static arguments through a constructor. These static arguments represent passing in a yet-to-be fired “finish” node for the FFT stage. Each stage is an ActivationAtom which is

constructed knowing the address of the next stage. When it fires, it sends out a new workload of ActivationAtoms set to output to the next stage. The FFT stage ActivationAtom is generic, and simply does a different combination of multiplications based on what stage number the system is in. Because each stage needs to be finished in order this is not a race condition.

While the runtime does not seamlessly allow implementing an N-point FFT, it does allow a constant resolution FFT. Many applications use such FFTs because the loops can be unrolled in the compiler, or recursion can be resolved, allowing for faster performance.

7.1.6 Summary

The FFT is a powerful, widely used algorithm that could largely benefit from a fine-grained execution model. Through our analysis, it has been shown that current efforts towards parallelization at a coarse-grain level provide substantially less parallelization opportunity than a fine-grained approach for the FFT. Current CPU architectures are not designed to take advantage of the huge amount of parallelism offered in this example. Therefore, it should be of interest to hardware architects of Digital Signal Processors to investigate techniques used by stream-based processors for future performance increases.

7.2 Quicksort Algorithm

Quicksort is a recursive sorting algorithm that works by picking an element of the list, organizing the list into elements greater than and less than this particular element, and repeating the operation on each side of that element until it reaches lists of length 1, and recombining them. Quicksort lends itself to parallelization because it is a divide and conquer algorithm. It splits the problem into sections; after each sorting operation, it calls itself again twice, so each new operation could be assigned to a new processor. This is of further interest because the type of parallelism used involves recursion rather than for loops.

```
quicksort(int[] arr)
{
	Int[] left, right;
	If(arr.length()<=1)
		Return arr;
	Pivot=removeAndReturnRandomElement(arr);
	For (elements elem in arr)
{
	If(elem<=Pivot)
		Left.add(elem);
	Else
		Right.add(elem);
}
Return Left.add(Pivot).add(Right);
}
```

Program 7.3 – Psuedo-code representing Quicksort

The algorithmic complexity for a normal Quicksort is O(n log(n)), because there are n comparisons made on each of log(n) levels of recursion. The computation could conceivably be parallelized in a way that dispatched each recursive call to be executed in parallel. Depending on the number of processors, instead of n operations on each level there would be $\frac{n}{2^i}$ (i being the level of

recursion) operations, and so a hypothetical computation time of $n * \sum_{i=1}^{\log(n)} \left(\frac{1}{2^i}\right)$. Since this sum is always less than 1, given unlimited processors the algorithm would run on average in a little less than O (n) time. With a limited number of processors, the complexity would be something more like

$n * \sum_{i=1}^{\log(p)} \left(\frac{1}{2^i}\right) + \left(\frac{n}{2^{\log(p)}}\right) * (\log(n) - \log(p))$; the same until it runs out of new processors to allocate to, plus the time for the remaining tree of required operations (on average the same across processors, done in parallel).

$$(x + a)^n = \sum_{i=1}^{\log(n)}$$

If a unit of time is defined as a single organization of a list around a pivot, then the expected time to complete a sort based on the number of processors available and the number of elements in the list can be determined using the formula above. It was done here with a graphing calculator:

Table 7.1 - 16 Processor Performance

Elements	Time	Parallel Time
500	1349	574
1000	3000	1279
10000	40000	17135
100000	500000	214754
1000000	6000000	2581575

Table 7.2 - 4 Processor Performance

Elements	Time	Parallel time
500	1349	690
1000	3000	1579
10000	40000	22368
100000	500000	289741
1000000	6000000	3556230

7.2.1 Front end implementation

The following diagram represents the Quicksort program. Its functionality is similar to the pseudo code algorithm shown above; a recursive function that sorts the contents of an array into two separate arrays and recombines them. The absence of modifiable variables in SISAL is handled by the way while-loops work- for the lists that are modified, a modified list is created and passed to the next iteration of the loop.

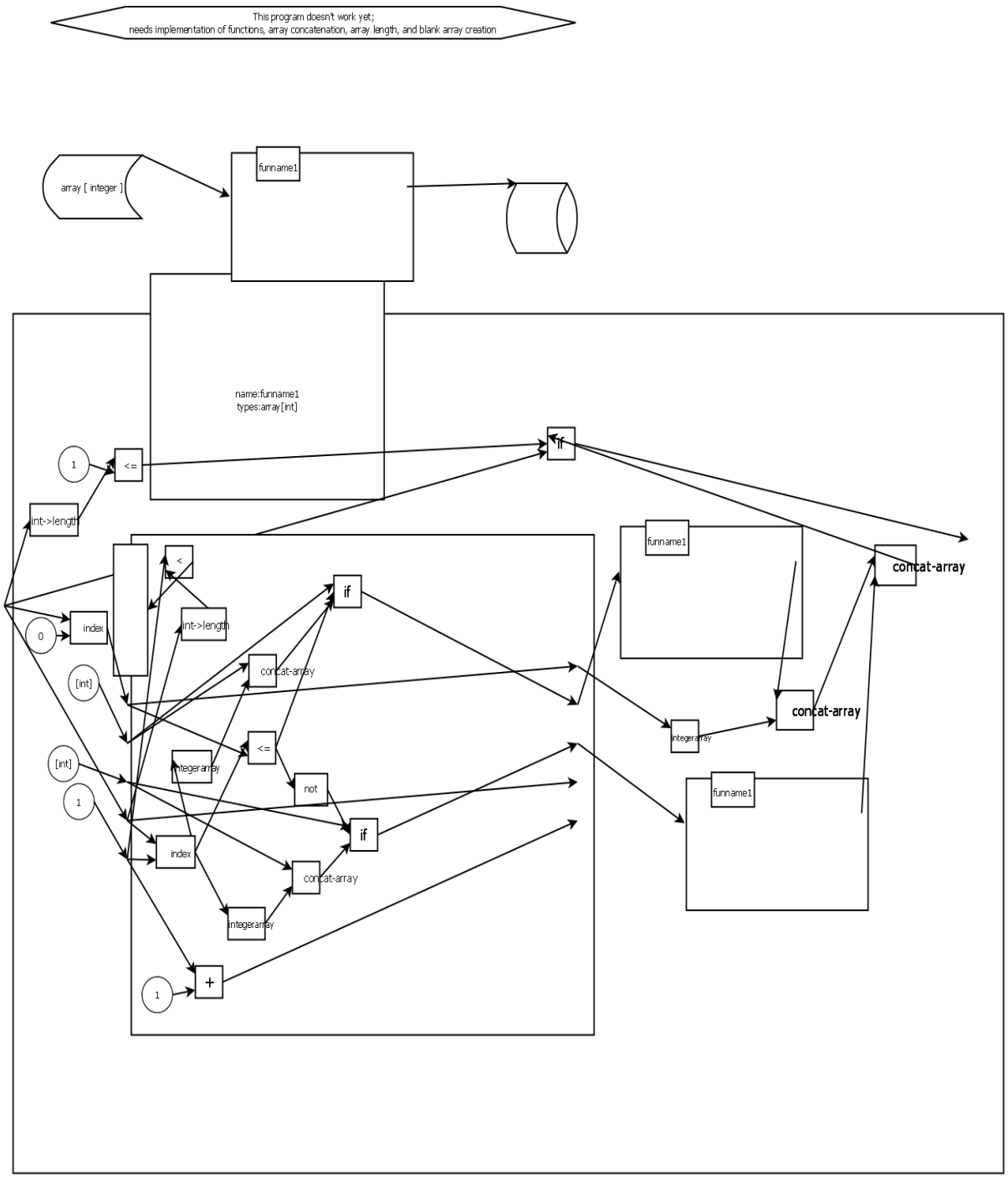

Figure 7.4 - Graphical Representation of Quicksort

7.3 H.264

The H.264 codec is a standard for video encoding and decoding that was originally introduced in 2003 [40]. It was conceived as the next step in Advanced Video Coding (AVC) and it was targeted at doubling the bit rate efficiency of previous codecs.

It should be noted however that "the H.264 draft standard does not explicitly define a CODEC (enCOder / DECoder pair). Rather, the standard defines the syntax of an encoded video bit-stream together with the method of decoding this bit-stream… there is [still] scope for considerable variation in the structure of the CODEC. The basic functional elements (prediction, transform, quantization, entropy encoding) are little different from previous standards (MPEG1, MPEG2, MPEG4, H.261, H.263); the important changes in H.264 occur in the details of each functional element." [40]

Over the past few years, the H.264 coding algorithm has been adopted in a wide range of applications, such as in YouTube's videos, iTunes' movies, and even in some Blu-Ray disks. We wanted to evaluate how well we could run a complicated CPU intensive algorithm on our runtime, and we chose this algorithm because of its increasing popularity as a video standard in the industry, as well as its openness in design and conduciveness to parallelization. To this end, we define the general implementation of the H.264 standard, and attempt to define and analyze a more parallel version at both the course and fine-grained level to be run against our runtime.

7.3.1 Implementation

7.3.1.1 Encoding

7.3.1.1.1 Brief Overview

There are two distinct dataflow paths for encoding a video, which can been seen in Figure 7.5. These are referred to as the "**Forward Pass**" and the "**Reconstruction Path**". The forward pass is where all of the video encoding occurs for each frame, and is represented by the blue dataflow path. The forward pass starts with the current frame of a video (F_n) and flows along the blue data flow path, whereas the reconstruction path begins with a previously encoded frame, or the reference frame(s) (F'_{n-1}), and flows along the pink data flow path. It is during the reconstruction phase that partially encoded frames are reconstructed so that the encoding process has end-to-end verifiability, proving that the encoded frames can be decoded.

While the primary encoding occurs in the forward pass, the reconstruction path is necessary in order for the encoder to "ensure that both the encoder and decoder use identical reference frames for intra prediction. This avoids possible encoder – decoder mismatches". [41]

In encoding new frames, and in reconstructing them, the dataflow diagram shows some inherent parallelism between the two operations, in that the forward pass hands off a copy of its frame to the reconstruction pass which can then be worked on simultaneously. Furthermore, because each frame is encoded individually (except for the reconstruction path which requires a previous set of

frames) we can consider encoding each frame in parallel. However, as the reader may see, the encoding operation itself is mostly sequential in design, and we address this further on.

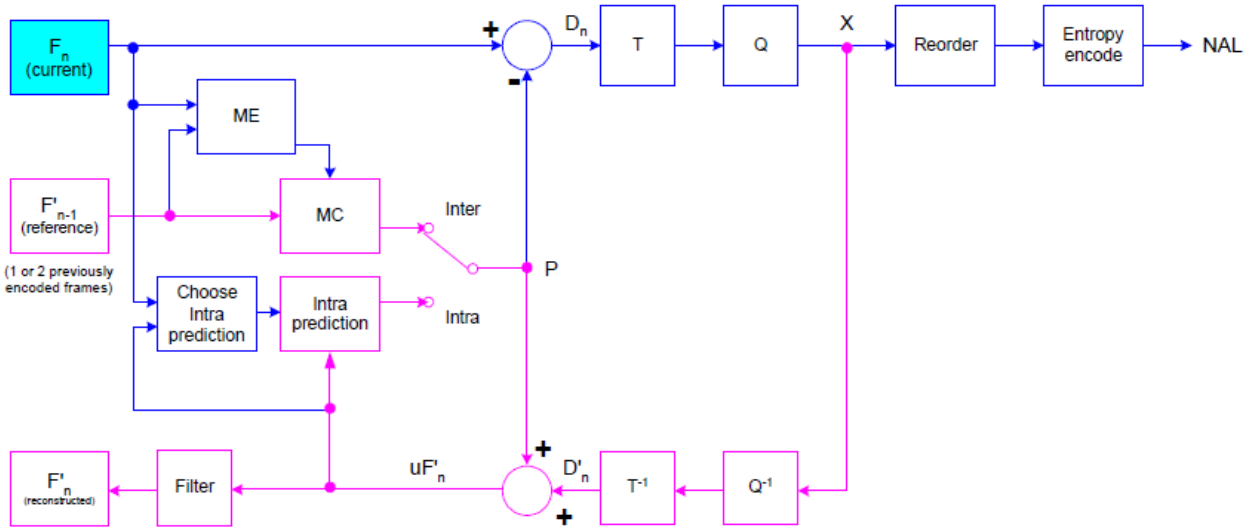

Figure 7.5 - Encoding Dataflow Diagram [40]

7.3.1.1.2 Forward Pass

The encoding process starts by taking in an **input frame** (F_n). This frame is then processed in batches of 'macro-blocks', which are simply blocks of 16 by 16 pixels taken from the original image. Each of these macro-blocks is then encoded via either **intra** or **inter** prediction, where a special prediction macro-block (P) is created from a reconstructed frame.

During **Inter** prediction, P is created via 'motion-compensated prediction' (MC) from a single, or multiple reference frames (F'_{n-1}). In **Intra** prediction, P is created from samples from the current frame (n) that have already been encoded, decoded and reconstructed.

Subsequently, a residual or 'difference' macro-block (D_n) is created by subtracting the prediction P from the current macro-block. This new macro-block is then **transformed** and **quantized** resulting in a new set of 'quantized transform coefficients' (X). This set of coefficients is then re-ordered and 'entropy encoded'.

In the next step, the **entropy-encoded** coefficients are merged with any further required data (such as the type of prediction used, the quantization step size etc) in order to form the final bit-stream in its compressed state. This bit-stream is then passed off to the Network Abstraction Layer (NAL), which is simply the terminal output node.

7.3.1.1.3 Reconstruction Path

In order to reconstruct a frame for encoding new macro-blocks, the quantized macro-block coefficients (X) must be decoded. These coefficients are re-scaled (Q^{-1}) and inverse transformed (T^{-1}), producing a new difference macro-block (D'_n). This new difference macro-block differs from the original

(D_n) in that during the quantization process, some data loss occurs, and so $D_n^{'}$ represents a distorted version of D_n.

In order to reconstruct the original macro-block, the prediction macro-block P is added to $D_n^{'}$, which results in $uF_n^{'}$, the unfiltered approximation of the original macro-block. Finally, to reduce the amount of distortion caused by blocking, a filter is applied to a series of these unfiltered frames which results in the final reconstructed reference frame.

7.3.1.2 Decoding

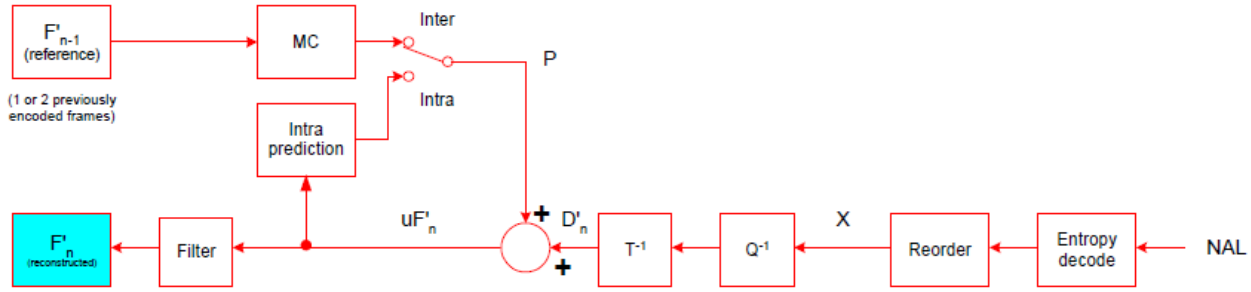

Figure 7.6 - Decoding Dataflow [40]

The decoder is very similar to the reconstruction path in the encoder, and it begins by receiving a stream of compressed bytes from the input node (NAL) as seen in Figure 7.6. The elements are then decoded via **entropy-decoding** and are reordered to create a new set of quantized coefficients (X). These then undergo a rescaling and are inverse transformed to produce the difference macro-block ($D_n^{'}$), which is identical to that found in the Encoder. Next, the prediction macro-block (P) is created using the metadata that was encoded in the bit-stream (again equivalent to that found in the Encoder). Finally, P is added (as opposed to subtracted) to $D_n^{'}$ to create the unfiltered frame ($uF_n^{'}$), which is then filtered to produce the decoded macro-block ($F_n^{'}$).

7.3.2 Fine-Grain Analysis

In order to fully take advantage of our fine-grained runtime, we need to make the initial program as parallel as possible. Taking a look at the encoding dataflow diagram in Figure , it may not be initially clear that we can gain any further parallelism considering that the reconstruction pass relies on previous iterations of the forward pass.

However, we can take advantage of further parallel optimizations by first optimizing our dataflow model, by breaking the loop around the Encoder into sets of two frame encoding passes and queuing up each type of pass for completion (forward pass, and reconstruction pass).

That is, instead of sequentially encoding and reconstructing each frame, one by one, we can encode all of the frames at a time, so that we can encode each frame (F_n) and then reconstruct it using (F_{n-1}) once each encoding pass is complete.

The hypothetical speedup that we can achieve with a course-grained model represented at the functional level with our runtime reaches a not-insignificant linear speedup as in Equation 7.4, where n is the number of frames to be encoded.

$$Concurrency\ Factor = \frac{1}{2}(n-1)$$

Equation 7.4 - Course-Grained Speedup Factor

However, further speedup is not only possible, but is expected given the use of our runtime and the fine-grained optimizations that can occur within each function itself. Since there are many different versions of the H.264 algorithm, with different implementations for each step in the encoding and decoding process [40], there is no single definitive methodology for calculating the maximum potential concurrency available in our runtime. However, we can make an attempt at an amortized fine-grained analysis of the encoding and decoding functions by reducing the functions into some basic variables based on their shared characteristics; namely the fact that each function mutates batches of pixels in some sequential order.

In our case, since our runtime queues up concurrent actions using ActivationAtoms at the primitive operation level, we can hypothetically deduce that the maximum potential speedup given the use of our runtime model grows exponentially as the number of possible concurrent operations grows. As it is the case that each function during both the encoding and decoding processes works with video frames, and that these frames are broken up into macro-blocks consisting of matrices of pixels, each operation on each pixel – or batch of pixels – can be run in parallel, since the operations at that level do not directly rely upon one another. In this case, the amount of parallelism that we can reasonably achieve is reflected by Equation 7.5 and Equation 7.6 where **N** is the number of frames, **b** is the number of pixel blocks that can be processed in parallel, and **o** is the number of operations that are performed on each block.

$$P = \frac{1}{2}(N-1) * \sum_{|P|} P'$$

Equation 7.5 - Total Fine-Grained Parallelism

$$P' = b * o$$

Equation 7.6 - Fine-Grained Parallelism per function

In practice however, significant speedup will only be observed on hardware that can run very many atomic operations at a time.

Other implementations of parallel H.264 encoding can be found with further reading such as with the MBRP parallel algorithm [42]; however they do not take advantage of the built-in parallelism in our runtime.

7.4 Summary of Results & Analysis

The hypothesis of this project was that graphical programming would be helpful in terms of parallel computing. Fine-grain parallelism was hypothesized to extract more parallelism out of a given computation than coarse-grain parallelism. While there was not enough analysis to fully confirm either hypothesis, our research indicates both are solid hypothesis, but yet to be fully proven. The system as a whole reached a level of construction sufficient for us to analyze the effectiveness of graphical programming and fine-grain parallelism. The goal of the graphical frontend was to convert a visual representation to a textual dataflow language. This goal was met because non-trivial, real-life problems were able to be expressed in this graphical notation. Furthermore, the runtime was complete enough for us to analyze real-life issues in terms of an implemented fine-grain parallelism system.

Through analyzing the Fast Fourier Transform we learned the idea of fine grain parallelism is a valid approach to extract parallelism from a computation. The FFT was able to achieve a massive increase in parallelism using fine-grain parallelism over a coarse-grain approach. A graphical representation of the FFT helped discover this parallelism, and our runtime could be used to implement the approach. Overall, the FFT proved to be a great example to exemplify the usefulness of both fine-grain parallelism and graphical programming.

H.264 is a more complicated algorithm than the FFT. Graphical representation in a dataflow form was shown to be helpful to extract parallelism. Parallelism is able to be exploited in this algorithm in a coarse-grain sense in order to maintain good performance on current machines. Fine-grain efforts could be implemented, but performance increases would only be seen on architectures that allow many atomic operations at a time. Overall, H.264 is another example of an algorithm which shows the usefulness of fine-grain parallelism & graphical programming.

Quicksort is a common sorting algorithm that is known to have parallel implementations. Graphical representations of the algorithm are helpful to determine parallelization opportunities. Because Quicksort requires information about its neighbors and iterations are not “self-contained” like the previous algorithms, a fine-grain implementation could not be derived. Thus, Quicksort is a good example showing the aide of graphical programming for parallel tasks.

Overall, the real life examples were good case studies confirming our project was useful research. All three of our case studies showed graphical representations seemed to help understand parallel opportunities. While some of these representations were not the representation we created, it still shows the promise of graphical programming techniques. Lastly, fine-grain parallelism seems to show promise in increasing parallelism and thus performance on architectures that support massive parallelism.

8 Conclusion

8.1 Lessons Learned

Initially in our investigation of data flow languages we encountered SISAL a language that seemed to address many of the issues that we were presented with in our MQP. By taking advantage of immutable, single assignment variables SISAL is able to trace data movement through a functional program. Additionally the pipelined compilation process in which SISAL is first translated into a intermediate graph and then into C code seemed to provide us with a chance to insert our own code without needing to change the outdated and undocumented bulk of the SISAL compiler. In hindsight we initially underestimated the size of two major shortcomings of SISAL and IF1. From this underestimation we have learned can be applied to future projects addressing the issues of implicit fine-grain parallelism.

8.1.1 The Age of SISAL

SISAL and IF1 were created in 1985; the age of the language has left us with little to no knowledge base or documentation for the language. We were able to obtain copy of IF1 documentation from former SISAL developer Pat Miller. Although at times this document was unclear, missing pages or out of date, it was our only resource for IF1 and made the work we did possible.

Future projects should focus on active open source projects as bases for development. Active projects often have a community that is willing to help developers working with project's code. In the event that some documentation is missing or unclear having this knowledge base can be a lifesaver.

8.1.2 Differences in Runtime

The biggest challenge in translating IF1 to our C++ runtime is in the differences between the two. These differences meant that the IF1 needed to be processed before it could be compiled into C++. Having not initially realized the extent of the task, our code generator was poorly structured and has become hard to maintain and debug.

Our runtime model is relatively novel and fully capable. In the future selecting a language that fits this model out of the box would be a better choice and would save considerable time in the implementation.

8.1.3 Further Lessons

Having a 6-person team caused addressable issues in terms of teamwork and collaboration. Underestimating the scope of the project initially has been a blessing to this end because it has given us the chance to break into small teams and focus our efforts. However there was not enough emphasis on team organization, structure and inter-team communication. Ultimately, having a leader appointed to each section of the project, and encouraging better design and organization principles would have improved the decision making process and the quality of the project as a whole.

8.2 Future Work

Our MQP has only begun to study the ability of a compiler to exploit implicit and fine-grained parallelism and the ability of a graphical front-end to aid in parallel programming. Through our research it has

become apparent that future work exists in further developing the abilities of our runtime model, code generation, and graphical programming approach.

8.2.1 Runtime

Future work could entail writing native compilers, which use our concept of activation atoms as a target. The output from this need not be architecture specific, but could utilize a virtual machine such as LLVM.

The runtime does not have a built-in implementation of loops and condition statements. The code generator uses a workaround to implement these concepts, which isn't the most efficient or straightforward way. Ideally the runtime would provide the tools necessary to implement these concepts, which is a better architectural design. Solving this next iteration of problems would enhance the performance and cohesion of our runtime environment.

8.2.2 Code Generation

The code generator for this project had the role of generating code from IF1 code that uses the fine-grain runtime. The IF1 code was generated from SISAL during which optimization such as inline function expansion and common sub-expression elimination are applied [43]. The current code generator performs no further optimizations to the code. This leaves room for improvements.

Further work could also be done on determining the best size in which to design execution blocks. Currently the code generator generates code that designates each operation as its own block of executable code. This means that every operation performed may be executed on a separate thread. Although this generates code that is highly parallelizable; on sequential architectures with only a few cores it is inefficient. Thus a valid open research question is determining the optimal size of a fine-grain execution block.

8.2.3 Graphical Front-End

Future work could transition the prototype graphical front end to an application specifically designed for graphical programming. The goal would be to improve the user and developer experience by tailoring the tool to the job of creating graphical dataflow programs, rather than using the current generic figure creation tool we used for prototyping.

While we found use of graphical representation useful for parallel computing, there was no wide study to confirm its usefulness. It may be an interesting study to interview a wide array of programmers on their opinion on using our tool or another graphical programming tool.

8.3 Conclusion

Ultimately, fine-grain parallelism and graphical programming show promise. The background research of this project highlights the importance of increasing the ease of parallel programming; without parallel computing performance gains will be minimal in future software. We successfully implemented a prototype system that makes use of a graphical programming language to generate code that utilizes the fine-grain runtime. Through this, we were able to compare real-life computing problems to a specific implementation of graphical programming and fine-grain parallelism. Using this system and

subsequent evaluation we were able to arrive at the conclusion that further research and investigation of both graphical programming and fine-grain parallelism would be useful.

9 Works Cited

[1] Asanovic, Krste, et al., The Landscape of Parallel Computing Research: A View from Berkeley, *EECS at UC Berkeley,* 12 18, 2006, retrieved 3 8, 2010 http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf.

[2] Computer History Museum, "Timeline of Computer History - 1959," *Computer History Museum,* Computer History Museum, retrieved 3 1, 2010, http://www.computerhistory.org/timeline/?year=1959.

[3] Bin Muhammad, Rashid, "History of Operating Systems," retrieved 3 1, 2010, http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/osHistory.htm.

[4] Computer History Museum, "Timeline of Computer History - 1962," *Computer History Museum,* Computer History Museum, retrieved 3 1, 2010, http://www.computerhistory.org/timeline/?year=1962.

[5] Intel Corporation, Intel Hyper-Threading Technology Technical User's Guide, 1 2003, retrieved 3 1, 2010 http://cache-www.intel.com/cd/00/00/01/77/17705_htt_user_guide.pdf.

[6] Gaudet, Dean, Process Model Design, 11 30, 1997, retrieved 3 2, 2010 http://arctic.org/~dean/apache/2.0/process-model.html.

[7] Barney, Blaise, "OpenMP," 1 30, 2009, retrieved 2 28, 2010, https://computing.llnl.gov/tutorials/openMP/.

[8] Hennessy, John L and Patterson, David A, *Computer architecture: a quantitative approach,* San Francisco : Morgan Kaufmann, 2007. p. 202,

[9] Patterson, David A. and Hennessy, John L., *Computer Organization and Design,* s.l. : Morgan Kaufmann, 2009.

[10] National Instruments Corporation, "Why Dataflow Programming Languages are Ideal for Programming Parallel Hardware," *National Instruments Developer Zone,* 11 1, 2007, retrieved 2 20, 2010, http://zone.ni.com/devzone/cda/tut/p/id/6098.

[11] Bode, Arndt, Eichele, Herbert and Pochayevets, Oleksandr, "BMDFM Official Site," retrieved 2 24, 2010, http://www.bmdfm.com/home.html.

[12] Lewis, Ted, "Prograph CPX - A Tutorial," *MacTech,* retrieved 3 8, 2010, http://www.mactech.com/articles/mactech/Vol.10/10.11/PrographCPXTutorial/index.html.

[13] Andescotia Software, "Marten 1.4," *Andescotia Software,* 2008, retrieved 2 24, 2010, http://www.andescotia.com/products/marten/.

[14] "The Mozart Programming System," retrieved 2 24, 2010, http://www.mozart-oz.org/.

[15] The MathWorks, Inc., "Simulink - Simulation and Model-Based Design," *The MathWorks,* retrieved 2 24, 2010, http://www.mathworks.com/products/simulink/.

[16] Ziring, Neal, "SISAL," *Dictionary of Programming Languages,* 1 2000, retrieved 3 7, 2010, http://cgibin.erols.com/ziring/cgi-bin/cep/cep.pl?_key=SISAL.

[17] Miller, Pat, "SISAL Parallel Programming," *Sourceforge,* retrieved 2 24, 2010, http://sourceforge.net/projects/sisal/.

[18] Ziring, Neal, VHDL, *Dictionary of Programming Languages,* 1 2000, retrieved 3 10, 2010 http://cgibin.erols.com/ziring/cgi-bin/cep/cep.pl?_key=VHDL.

[19] Institute of Electronic Music and Acoustics, PD Community Site, *Pure Data,* 11 25, 2009, retrieved 3 10, 2010 http://puredata.info/.

[20] Mitov Software, OpenWire, *Mitov Software,* retrieved 3 10, 2010 http://www.mitov.com/html/openwire.html.

[21] Cycling '74, Max, *Cycling '74,* retrieved 3 10, 2010 http://cycling74.com/products/maxmspjitter/.

[22] "Cantata," http://greta.cs.ioc.ee/~khoros2/k2tools/cantata/cantata.html.

[23] "jMax," http://freesoftware.ircam.fr/rubrique.php3?id_rubrique=14.

[24] Ziring, Neal, "Lucid," *Dictionary of Programming Languages,* 1 2000, retrieved 2 28, 2010, http://cgibin.erols.com/ziring/cgi-bin/cep/cep.pl?_key=Lucid.

[25] National Instruments Corporation, "NI LabVIEW - The Software that Powers Virtual Instrumentation," *National Instruments,* National Instruments Corporation, retrieved 2 28, 2010, http://www.ni.com/labview/.

[26] National Instruments Corporation, "LabVIEW 2009 Features," *National Instruments,* National Instruments Corporation, retrieved 2 28, 2010, http://www.ni.com/labview/whatsnew/features.htm.

[27] Miller, Pat, "Sisal Lives!" retrieved 2 20, 2010, http://sisal.sourceforge.net/.

[28] Hollingsworth, Karen, SISAL: Streams and Iteration in a Single Assignment Language, 10 10, 2006.

[29] Cann, David C., Retire Fortran? A Debate Rekindled, Livermore, California, United States of America : Lawrence Livermore National Laboratory, 7 24, 1991.

[30] Cann, David C., The Optimizing SISAL Compiler: Version 12.0, Livermore, California, United States of America : Lawrence Livermore National Laboratory, 1992.

[31] Skedzielewski, Stephen and Glauert, John, IF1: An Intermediate Form for Applicative Languages, Livermore, California, United States of America : Lawrence Livermore National Laboratory, 7 31, 1985.

[32] "Intel Thread Building Blocks 2.2," http://www.threadingbuildingblocks.org/.

[33] "Hadoop," http://hadoop.apache.org/.

[34] Dean, Jeffrey and Ghemawat, Sanjay, MapReduce: Simplified Data Processing on Large Clusters, 2004.

[35] "Project Fortress," http://projectfortress.sun.com/Projects/Community.

[36] Adve, Vikram, et al., 1998. SC'98: High Performance Computing and Networking,

[37] "Graphviz," http://www.graphviz.org/.

[38] Lyons, Richard, *Understanding Digital Signal Processing,* s.l. : Pearson Education, 1st Edition, 1996.

[39] Moreland, Kenneth and Angel, Edward, s.l. : SIGGRAPH/Eurographics Workshop on Graphics Hardware 2003 Proceedings, 2003.

[40] Richardson, Iain E G, H.264 / MPEG-4 Parth 10 White Paper, *vcodex,* August 10, 2002, www.vcodex.com.

[41] Hamzaoglu, Ilker, Tasdizen, Ozgur and Sahin, Esra, AN EFFICIENT H.264 INTRA FRAME CODER SYSTEM DESIGN, *Sabanci University,* 2007, http://people.sabanciuniv.edu/~hamzaoglu/papers/vlsisoc07_intra.pdf.

[42] Sun, Shuwei, Wamg, Dong and Chen, Shuming, A Highly Efficient Parallel Algorithm for H.264, *National University of Defense Technology,* 2007, http://www2.cs.uh.edu/~openuh/hpcc07/papers/55-Sun.pdf.

[43] *Data Flow Graph Optimization in IF1,* Nancy, France : s.n., 1987.

[44] "Dia," http://live.gnome.org/Dia.